# Table of Contents

1. Introduction .................................................................................................................. 2
   1.1. What is Cypher? .................................................................................................. 2
2. Syntax ............................................................................................................................... 13
   2.1. Values and types ................................................................................................. 15
   2.2. Naming rules and recommendations .................................................................. 17
   2.3. Expressions ......................................................................................................... 19
   2.4. Variables .............................................................................................................. 24
   2.5. Reserved keywords ............................................................................................. 25
   2.6. Parameters .......................................................................................................... 28
   2.7. Operators ............................................................................................................ 32
3. Using a regular expression with \[=\] to filter words ..................................................... 43
4. Boolean operators .......................................................................................................... 44
   4.1. Using boolean operators to filter numbers ....................................................... 44
5. String operators ............................................................................................................ 45
   5.1. Concatenating two strings with + ........................................................................ 45
6. Temporal operators ....................................................................................................... 46
   6.1. Adding and subtracting a Duration to or from a temporal instant .................... 46
   6.2. Adding and subtracting a Duration to or from another Duration .................... 47
   6.3. Multiplying and dividing a Duration with or by a number .................................. 48
7. Map operators ............................................................................................................... 49
   7.1. Statically accessing the value of a nested map by key using the \[\] operator ....... 49
   7.2. Dynamically accessing the value of a map by key using the \[\] operator and a parameter ........................................................................................................ 49
8. List operators ................................................................................................................ 50
   8.1. Concatenating two lists using + .......................................................................... 50
   8.2. Using IN to check if a number is in a list .......................................................... 50
   8.3. Using IN for more complex list membership operations ................................... 50
   8.4. Accessing elements in a list using the \[\] operator .............................................. 51
   8.5. Dynamically accessing an element in a list using the \[\] operator and a parameter ........................................................................................................ 52
   8.6. Using IN with \[\] on a nested list ....................................................................... 52
   8.7. Comments .......................................................................................................... 53
   8.8. Patterns .............................................................................................................. 53
   8.9. Temporal (Date/Time) values .............................................................................. 58
   8.10. Spatial values .................................................................................................... 76
   8.11. Lists .................................................................................................................. 82
This is the Cypher manual for Neo4j version 4.4, authored by the Neo4j Team.

This manual covers the following areas:

- **Introduction** — Introducing the Cypher query language.
- **Syntax** — Learn Cypher query syntax.
- **Clauses** — Reference of Cypher query clauses.
- **Functions** — Reference of Cypher query functions.
- **Indexes for search performance** — How to manage indexes used for search performance.
- **Full-text search index** — How to use full-text indexes, to enable full-text search.
- **Constraints** — How to manage constraints used for ensuring data integrity.
- **Database management** — How to use Cypher to manage Neo4j databases.
- **Database alias management** — How to use Cypher to manage database aliases in Neo4j.
- **Access control** — How to manage Neo4j role-based access control and fine-grained security.
- **Query tuning** — Learn to analyze queries and tune them for performance.
- **Execution plans** — Cypher execution plans and operators.
- **Deprecations, additions and compatibility** — An overview of language developments across versions.
- **Glossary of keywords** — A glossary of Cypher keywords, with links to other parts of the Cypher manual.
- **Cypher styleguide** — A guide to the recommended style for writing Cypher queries.

Who should read this?

This manual is written for the developer of a Neo4j client application.
Chapter 1. Introduction

This section provides an introduction to the Cypher query language.

1.1. What is Cypher?

Cypher is a declarative graph query language that allows for expressive and efficient querying, updating and administering of the graph. It is designed to be suitable for both developers and operations professionals. Cypher is designed to be simple, yet powerful; highly complicated database queries can be easily expressed, enabling you to focus on your domain, instead of getting lost in database access.

Cypher is inspired by a number of different approaches and builds on established practices for expressive querying. Many of the keywords, such as WHERE and ORDER BY, are inspired by SQL. Pattern matching borrows expression approaches from SPARQL. Some of the list semantics are borrowed from languages such as Haskell and Python. Cypher's constructs, based on English prose and neat iconography, make queries easy, both to write and to read.

Names are case-sensitive. :PERSON, :Person and :person are three different labels, as well as n and N are two different variables.

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. These are discussed in more detail in the chapter on Clauses.

The following are a few examples of 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.

Let’s create a simple example graph with the following query:

```cypher
CREATE (john:Person {name: 'John'})
CREATE (joe:Person {name: 'Joe'})
CREATE (steve:Person {name: 'Steve'})
CREATE (sara:Person {name: 'Sara'})
CREATE (maria:Person {name: 'Maria'})
CREATE (john)-[:FRIEND]->(joe)-[:FRIEND]->(steve)
CREATE (john)-[:FRIEND]->(sara)-[:FRIEND]->(maria)
```
Example Graph

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

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

Resulting in:

```
+----------------------+
| john.name | fof.name |
+----------------------+
| "John"    | "Maria"  |
| "John"    | "Steve"  |
+----------------------+
2 rows
```

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

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

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

Resulting in:

```
+---------------------------+
| user.name | follower.name |
+---------------------------+
| "John"    | "Sara"        |
| "Joe"     | "Steve"       |
+---------------------------+
2 rows
```

And these 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.
1.2. Neo4j databases and graphs

This section describes databases and graphs in Neo4j.

Cypher queries are executed against a Neo4j database, but normally apply to specific graphs. It is important to understand the meaning of these terms and exactly when a graph is not a database.

DBMS

A Neo4j Database Management System is capable of containing and managing multiple graphs contained in databases. Client applications will connect to the DBMS and open sessions against it. A client session provides access to any graph in the DBMS.

Graph

This is a data model within a database. Normally there is only one graph within each database, and many administrative commands that refer to a specific graph do so using the database name.

Cypher queries executed in a session may declare which graph they apply to, or use a default, given by the session.

In Neo4j Fabric it is possible to refer to multiple graphs within the same query.

Database

A database is a storage and retrieval mechanism for collecting data in a defined space on disk and in memory.

Most of the time Cypher queries are reading or updating queries, which are run against a graph. There are also administrative commands that apply to a database, or to the entire DBMS. Administrative commands cannot be run in a session connected to a normal user database, but instead need to be run within a session connected to the system database.

1.2.1. The system database and the default database

All Neo4j servers contain a built-in database called system, which behaves differently than all other databases. The system database stores system data and you can not perform graph queries against it.

A fresh installation of Neo4j includes two databases:

- system - the system database described above, containing meta-data on the DBMS and security configuration.
- neo4j - the default database, named using the config option dbms.default_database=neo4j.

For more information about the system database, see the sections on Database management and Access control.
1.2.2. Different editions of Neo4j

Neo4j has two editions, a commercial Enterprise Edition with additional performance and administrative features, and an open-source Community Edition. Cypher works almost identically between the two editions, and as such most of this manual will not differentiate between them. In the few cases where there is a difference in Cypher language support or behaviour between editions, these are highlighted as described below in Limited Support Features.

However it is worth listing up-front the key areas that are not supported in the open-source edition:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Enterprise</th>
<th>Community</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-database</td>
<td>Any number of user databases</td>
<td>Only system and one user database</td>
</tr>
<tr>
<td>Role-based security</td>
<td>User, role, and privilege management for flexible access control and sub-graph access control.</td>
<td>Multi-user management. All users have full access rights.</td>
</tr>
<tr>
<td>Constraints</td>
<td>Existence constraints, uniqueness constraints, and NODE KEY constraints.</td>
<td>Only uniqueness constraints</td>
</tr>
</tbody>
</table>

1.2.3. Limited Support Features

Some elements of Cypher do not work in all deployments of Neo4j, and we use specific markers to highlight these cases:

<table>
<thead>
<tr>
<th>Marker</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>deprecated</td>
<td>This feature is deprecated and will be removed in a future version</td>
<td>DROP INDEX ON :Label(property)</td>
</tr>
<tr>
<td>enterprise-only</td>
<td>This feature only works in the enterprise edition of Neo4j</td>
<td>CREATE DATABASE foo</td>
</tr>
<tr>
<td>fabric</td>
<td>This feature only works in a fabric deployment of Neo4j</td>
<td>USE fabric.graph(0)</td>
</tr>
</tbody>
</table>

1.3. Querying, updating and administering

This section describes using Cypher for both querying and updating your graph, as well as administering graphs and databases.

In the introduction we described the common case of using Cypher to perform read-only queries of the graph. However, it is also possible to use Cypher to perform updates to the graph, import data into the graph, and perform administrative actions on graphs, databases and the entire DBMS.

All these various options are described in more detail in later sections, but it is worth summarizing a few key points first.
1.3.1. The structure of administrative queries

Cypher administrative queries cannot be combined with normal reading and writing queries. Each administrative query will perform either an update action to the system or a read of status information from the system. Some administrative commands make changes to a specific database, and will therefore be possible to run only when connected to the database of interest. Others make changes to the state of the entire DBMS and can only be run against the special system database.

1.3.2. The structure of update queries

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.

A Cypher query part can either read and match on the graph, or make updates on it, not both simultaneously.

If your query only performs reads, Cypher will 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 is performed.

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.

1.3.3. Returning data

Any query can return data. If a query only reads, it has to return data. If a read-query doesn’t return any data, it serves no purpose, and is therefore 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 \texttt{RETURN} clause has three sub-clauses that come with it: \texttt{SKIP/LIMIT} and \texttt{ORDER BY}.

If you return nodes or relationships from a query that has just deleted them — beware, you are holding a pointer that is no longer valid.

1.4. Transactions

\textit{This section describes how} Cypher queries \textit{work with database transactions.}

All Cypher queries run within transactions. Modifications done by updating queries are held in memory by the transaction until it is committed, at which point the changes are persisted to disk and become visible to other transactions. If an error occurs - either during query evaluation, such as division by zero, or during commit, such as constraint violations - the transaction is automatically rolled back, and no changes are persisted in the graph.

In short, an updating query always either fully succeeds, or does not succeed at all.

| A query that makes a large number of updates consequently uses large amounts of memory since the transaction holds changes in memory. For memory configuration in Neo4j, see the Neo4j Operations Manual \(
\text{\texttt{\rightarrow Memory configuration.}}
\) |

Transactions can be either explicit or implicit.

- \textbf{Explicit transactions:}
  - Are opened by the user.
  - Can execute multiple Cypher queries in sequence.
  - Are committed, or rolled back, by the user.

- \textbf{Implicit transactions, sometimes called auto-commit transactions or :auto transactions:}
  - Are opened automatically.
  - Can execute a single Cypher query.
  - Are committed automatically when the query finishes successfully.

Queries that start separate transactions themselves, such as queries using \texttt{CALL \{ ... \} IN TRANSACTIONS} or \texttt{PERIODIC COMMIT} are only allowed in implicit mode.

For examples of the API’s used to start and commit transactions, refer to the API specific documentation:

- For information on using transactions with a Neo4j driver, see \texttt{The session API} in the \texttt{Neo4j Driver manuals}.
- For information on using transactions over the HTTP API, see the \texttt{HTTP API documentation \texttt{\rightarrow Using the HTTP API}}.
- For information on using transactions within the embedded Core API, see \texttt{Java Reference \texttt{\rightarrow Executing Cypher queries from Java}}.
When writing procedures or using Neo4j embedded, remember that all iterators returned from an execution result should be either fully exhausted or closed. This ensures that the resources bound to them are properly released.

### 1.4.1. DBMS Transactions

Beginning a transaction while connected to a DBMS will start a DBMS-level transaction. A DBMS-level transaction is a container for database transactions.

A database transaction is started when the first query to a specific database is issued. Database transactions opened inside a DBMS-level transaction are committed or rolled back when the DBMS-level transaction is committed or rolled back.

For an example of how queries to multiple databases can be issued in one transaction, see Databases and execution context in the Neo4j Driver manuals.

DBMS transactions have the following limitations:

- Only one database can be written to in a DBMS transaction
- Cypher operations fall into the following main categories:
  - Operations on graphs.
  - Schema commands.
  - Administration commands.

It is not possible to combine any of these workloads in a single DBMS transaction.

### 1.5. Cypher path matching

*Cypher path matching uses relationship isomorphism, the same relationship cannot be returned more than once in the same result record.*

Neo4j Cypher makes use of relationship isomorphism for path matching and is a very effective way of reducing the result set size and preventing infinite traversals.

In Neo4j, all relationships have a direction. However, you can have the notion of undirected relationships at query time.

In the case of variable length pattern expressions, it is particularly important to have a constraint check, or an infinite number of result records could be found.

To understand this better, let us consider a few alternative options:

**Homomorphism**

No constraints for path matching.
Node isomorphism
The same node cannot be returned more than once for each path matching record.

Relationship isomorphism
The same relationship cannot be returned more than once for each path matching record. Cypher makes use of relationship isomorphism for path matching.

1.5.1. Homomorphism
Constraints: No constraints for path matching.

Example 1. Homomorphism

The graph is composed of only two nodes \((a)\) and \((b)\), connected by one relationship, \((a:Node)\)\-(\[r:R]\)->\((b:Node)\).

If the query is looking for paths of length \(n\) and do not care about the direction, a path of length \(n\) will be returned repeating the two nodes over and over.

For example, find all paths with 5 relationships and do not care about the relationship direction:

MATCH p = ()-[*5]-(())
RETURN nodes(p)

This will return the two resulting records if homomorphism was used, \([a, b, a, b, a, b]\), as well as \([b, a, b, a, b, a]\).

1.5.2. Node isomorphism
Constraints: The same node cannot be returned more than once for each path matching record.

In another two-node example, such as \((a:Node)-[r:R]->(b:Node)\); only paths of length 1 can be found with the node isomorphism constraint.

Example 2. Node isomorphism

The graph is composed of only two nodes \((a)\) and \((b)\), connected by one relationship, \((a:Node)\)\-(\[r:R]\)->\((b:Node)\).

MATCH p = ()-[*1]-(())
RETURN nodes(p)

This will return the two resulting records if node isomorphism was used, \([a, b]\), as well as \([b, a]\).

1.5.3. Relationship isomorphism
Constraints: The same relationship cannot be returned more than once for each path matching record.
In another two-node example, such as \((a:Node)\-[r:R]\)->(b:Node); only paths of length 1 can be found with the relationship isomorphism constraint.

Example 3. Relationship isomorphism

The graph is composed of only two nodes \((a)\) and \((b)\), connected by one relationship, \((a:Node)\-[r:R]\->(b:Node)\).

```cypher
MATCH p = ()-[*1]->()
RETURN nodes(p)
```

This will return the two resulting records \([a, b]\), as well as \([b, a]\).

1.5.4. Cypher path matching example

Cypher makes use of relationship isomorphism for path matching.
Example 4. Friend of friends

Looking for a user’s friends of friends should not return said user.

To demonstrate this, let’s create a few nodes and relationships:

Query 1, create data.

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

Nodes created: 3
Relationships created: 2
Properties set: 3

Which gives us the following graph:

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

Query 2, friend of friends of Adam.

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

<table>
<thead>
<tr>
<th>fofName</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;David&quot;</td>
</tr>
</tbody>
</table>
Rows: 1

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:
Query 3, multiple MATCH clauses.

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

<table>
<thead>
<tr>
<th>fofName</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;David&quot;</td>
</tr>
<tr>
<td>&quot;Adam&quot;</td>
</tr>
</tbody>
</table>

Rows: 2

Note that while the following Query 4 looks similar to Query 3, it is actually equivalent to Query 2.

Query 4, equivalent to query 2.

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

<table>
<thead>
<tr>
<th>fofName</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;David&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

Here, the MATCH clause has a single pattern with two paths, while the previous query has two distinct patterns.
Chapter 2. 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
- Operators
  - Operators at a glance
  - Aggregation operators
  - Property operators
  - Mathematical operators
  - Comparison operators
  - Boolean operators
  - String operators
  - Temporal operators
  - Map operators
  - List operators
- 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
  ° 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
2.1. Values and types

This section provides an overview of data types in Cypher.

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

- Integer, Float, String, Boolean, Point, Date, Time, LocalTime, DateTime, LocalDateTime, and Duration.

Structural types

- Node, Relationship, and Path.

Composite types

- List and Map.

2.1.1. Property types

☑ Can be returned from Cypher queries
☑ Can be used as parameters
☑ Can be stored as properties
☑ Can be constructed with Cypher literals

The property types:

- **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.

<table>
<thead>
<tr>
<th>Sorting of special characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strings that contain characters that do not belong to the Basic Multilingual Plane (BMP) can have inconsistent or non-deterministic ordering in Neo4j. BMP is a subset of all characters defined in Unicode. Expressed simply, it contains all common characters from all common languages.</td>
</tr>
</tbody>
</table>

The most significant characters not in BMP are those belonging to the Supplementary Multilingual Plane or the Supplementary Ideographic Plane. Examples are:

- Historic scripts and symbols and notation used within certain fields such as: Egyptian hieroglyphs, modern musical notation, mathematical alphanumerics.
- Emojis and other pictographic sets.
- Game symbols for playing cards, Mah Jongg, and dominoes.
- CJK Ideograph that were not included in earlier character encoding standards.

2.1.2. Structural types

- Can be returned from Cypher queries
- Cannot be used as parameters
- Cannot be stored as properties
- Cannot be constructed with Cypher literals

The structural types:

- Node
  - Id
  - Label(s)

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

  - Map (of properties)

- Relationship
  - Id
Nodes, relationships, and paths are returned as a result of pattern matching. In Neo4j, all relationships have a direction. However, you can have the notion of undirected relationships at query time.

### 2.1.3. Composite types

- Can be returned from Cypher queries
- Can be used as parameters
- Cannot be stored as properties
- Can be constructed with Cypher literals

The composite types:

- **List**, a heterogeneous, ordered collection of values, each of which has any property, structural or composite type.

- **Map**, a heterogeneous, unordered collection of (Key, Value) pairs.
  - **Key** is a String
  - **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](#)).

### 2.2. Naming rules and recommendations

This section describes rules and recommendations for the naming of node labels, relationship types, property names, variables, indexes, and constraints.

#### 2.2.1. Naming rules

- **Alphabetic characters:**
  - Names should begin with an alphabetic character.
  - This includes "non-English" characters, such as å, ä, ö, ü etc.
Numbers:
- Names should not begin with a number.
- To illustrate, 1first is not allowed, whereas first1 is allowed.

Symbols:
- Names should not contain symbols, except for underscore, as in my_variable, or $ as the first character to denote a parameter, as given by $myParam.

Length:
- Can be very long, up to 65535 \( (2^{16} - 1) \) or 65534 characters, depending on the version of Neo4j.

Case-sensitive:
- Names are case-sensitive and 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.

Non-alphabetic characters, including numbers, symbols and whitespace characters, can be used in names, but must be escaped using backticks. For example: '^n', '1first', '$n', and 'my variable has spaces'. Database names are an exception and may include dots without the need for escaping. For example: naming a database foo.bar.baz is perfectly valid.

2.2.2. 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).

2.2.3. Recommendations

Here are the recommended naming conventions:

<table>
<thead>
<tr>
<th>Node labels</th>
<th>Camel-case, beginning with an uppercase character</th>
<th>:VehicleOwner rather than :vehicle_owner etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relationship types</td>
<td>Upper-case, using underscore to separate words</td>
<td>:OWNS_VEHICLE rather than :ownsVehicle etc.</td>
</tr>
</tbody>
</table>
2.3. Expressions

This section contains an overview of expressions in Cypher with examples.

- **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

### 2.3.1. Expressions in general

Most expressions in Cypher evaluate to **null** if any of their inner expressions are **null**. Notable exceptions are the operators **IS NULL** and **IS NOT NULL**.

An expression in Cypher can be:

- A decimal (integer or float) literal: 13, -40000, 3.14
- A decimal (integer or float) literal in scientific notation: 6.022E23.
- A hexadecimal integer literal (starting with **0x**): 0x13af, 0xFC3A9, -0x66eff.
- An octal integer literal (starting with **0o** or **0**): 0o1372, 02127, -0o5671.
- A string literal: 'Hello', "World".
- A boolean literal: 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)-[r]->(b), (a)-[r]-(b), (a)--(b), (a)-->()<--(b)\.
- An operator application: 1 + 2, 3 < 4\.
- A predicate expression is an expression that returns true or false: a.prop = 'Hello', length(p) > 10, a.name IS NOT NULL\.
- An existential subquery is an expression that returns true or false: EXISTS { MATCH (n)-[r]->(p) WHERE p.name = 'Sven' }\.
- A regular expression: a.name =~ 'Tim.*'.


2.3.2. Note on string literals

String literals can contain the following escape sequences:

<table>
<thead>
<tr>
<th>Escape sequence</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>\t</td>
<td>Tab</td>
</tr>
<tr>
<td>\b</td>
<td>Backspace</td>
</tr>
<tr>
<td>\n</td>
<td>Newline</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage return</td>
</tr>
<tr>
<td>\f</td>
<td>Form feed</td>
</tr>
<tr>
<td>'</td>
<td>Single quote</td>
</tr>
<tr>
<td>&quot;</td>
<td>Double quote</td>
</tr>
<tr>
<td>\.</td>
<td>Backslash</td>
</tr>
<tr>
<td>\uxxxx</td>
<td>Unicode UTF-16 code point (4 hex digits must follow the \u)</td>
</tr>
<tr>
<td>\Uxxxxxxxx</td>
<td>Unicode UTF-32 code point (8 hex digits must follow the \U)</td>
</tr>
</tbody>
</table>

2.3.3. CASE expressions

Generic conditional expressions may be expressed using the 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.

CASE can only be used as part of RETURN or WITH if you want to use the result in the succeeding clause or statement.

The following graph is used for the examples below:
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:**

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

**Arguments:**

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

**Query**

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

**Table 1. Result**

| Name   | Description |
|--------|-------------|-------------|
|        |             | 21          |
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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>predicate</td>
<td>A predicate that is tested to find a valid alternative.</td>
</tr>
<tr>
<td>result</td>
<td>This is the expression returned as output if <code>predicate</code> evaluates to <code>true</code>.</td>
</tr>
<tr>
<td>default</td>
<td>If no match is found, <code>default</code> is returned.</td>
</tr>
</tbody>
</table>

**Query**

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

**Table 2. Result**

<table>
<thead>
<tr>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
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**

```sql
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 3. Result**

<table>
<thead>
<tr>
<th>n.name</th>
<th>age_10_years_ago</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Alice&quot;</td>
<td>28</td>
</tr>
<tr>
<td>&quot;Bob&quot;</td>
<td>15</td>
</tr>
<tr>
<td>&quot;Charlie&quot;</td>
<td>43</td>
</tr>
<tr>
<td>&quot;Daniel&quot;</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;Eskil&quot;</td>
<td>31</td>
</tr>
</tbody>
</table>

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

**Query**

```sql
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 4. Result**

<table>
<thead>
<tr>
<th>n.name</th>
<th>age_10_years_ago</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Alice&quot;</td>
<td>28</td>
</tr>
<tr>
<td>&quot;Bob&quot;</td>
<td>15</td>
</tr>
</tbody>
</table>
Using the result of `CASE` in the succeeding clause or statement

You can use the result of `CASE` to set properties on a node or relationship. For example, instead of specifying the node directly, you can set a property for a node selected by an expression:

Query

```cypher
MATCH (n)
WITH n, 
CASE n.eyes
  WHEN 'blue' THEN 1
  WHEN 'brown' THEN 2
  ELSE 3
END AS colourCode
SET n.colourCode = colourCode
```

For more information about using the `SET` clause, see `SET`.

Table 5. Result

(usual result)

<table>
<thead>
<tr>
<th>Rows: 0</th>
<th>Properties set: 5</th>
</tr>
</thead>
</table>

2.4. Variables

This section provides an overview of variables in Cypher.

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:

```cypher
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`.

2.5. Reserved keywords

This section contains a list of reserved keywords in Cypher.

Reserved keywords are words that have a special meaning in Cypher. The listing of the reserved keywords are grouped by the categories from which they are drawn. In addition to this, there are a number of keywords that are reserved for future use.

The reserved keywords 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.

2.5.1. Clauses

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

2.5.2. Subclauses
• LIMIT
• ORDER
• SKIP
• WHERE
• YIELD

2.5.3. Modifiers
• ASC
• ASCENDING
• ASSERT
• BY
• CSV
• DESC
• DESCENDING
• ON

2.5.4. Expressions
• ALL
• CASE
• ELSE
• END
• THEN
• WHEN

2.5.5. Operators
• AND
• AS
• CONTAINS
• DISTINCT
• ENDS
• IN
• IS
• NOT
• OR
• STARTS
• XOR

2.5.6. Schema

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

2.5.7. Hints

• INDEX
• JOIN
• PERIODIC
• COMMIT
• SCAN
• USING

2.5.8. Literals

• false
• null
• true

2.5.9. Reserved for future use

• ADD
• DO
• FOR
• MANDATORY
• OF
• REQUIRE
2.6. Parameters

This section describes parameterized querying.

2.6.1. Introduction

Cypher supports querying with parameters. A parameterized query is a query in which placeholders are used for parameters and the parameter values are supplied at execution time. This means developers do not 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

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.

Setting parameters when running a query is dependent on the client environment. For example:

- To set a parameter in Cypher Shell use :param name => 'Joe'. For more information refer to Operations Manual → Cypher Shell - Query Parameters.
- For Neo4j Browser use the same syntax as Cypher Shell, :param name => 'Joe'.
- When using drivers, the syntax is dependent on the language choice. See the examples in Transactions in the Neo4j Driver manuals.
- For usage via the Neo4j HTTP API, see the HTTP API documentation.

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.

<table>
<thead>
<tr>
<th>Information</th>
</tr>
</thead>
</table>

The old parameter syntax `{param}` was deprecated in Neo4j 3.0 and removed entirely in Neo4j 4.0. Using it will result in a syntax error. However, it is still possible to use it, with warnings, if you prefix the query with `CYPHER 3.5`. See Cypher Compatibility for further information.
2.6.2. Auto-parameterization

When a query does not use parameters, Cypher will try to infer parameters anyway. Each literal in the query is replaced with a parameter. This increases the re-usability of the computed plan for queries that are identical except for the literals. It is not recommended to rely on this behavior - users should rather use parameters where they think it is appropriate.

If at least one parameter is used in the query, auto-parameterization is turned off for that query. This means that any remaining literals will not be turned into parameters.

2.6.3. String literal

Parameters

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

Query

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

You can use parameters in this syntax as well:

Parameters

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

Query

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

2.6.4. Regular expression

Parameters

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

Query

```cypher
MATCH (n:Person)
WHERE n.name =~ $regex
RETURN n.name
```
2.6.5. Case-sensitive string pattern matching

Parameters

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

Query

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

2.6.6. Create node with properties

Parameters

```json
{
  "props": {
    "name": "Andy",
    "position": "Developer"
  }
}
```

Query

```
CREATE ($props)
```

2.6.7. Create multiple nodes with properties

Parameters

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

Query

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

2.6.8. Setting all properties on a node

Note that this will replace all the current properties.
Parameters

```json
{
    "props": {
        "name": "Andy",
        "position": "Developer"
    }
}
```

Query

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

2.6.9. **SKIP** and **LIMIT**

Parameters

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

Query

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

2.6.10. **Node id**

Parameters

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

Query

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

2.6.11. **Multiple node ids**

Parameters

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

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

2.6.12. Calling procedures

Parameters

```json
{
  "indexname" : "My index"
}
```

Query

CALL db.resampleIndex($indexname)

2.7. Operators

This section contains an overview of operators.

- Operators at a glance
- Aggregation operators
  - Using the **DISTINCT** operator
- Property operators
  - Statically accessing a property of a node or relationship using the **.** operator
  - Filtering on a dynamically-computed property key using the **[ ]** operator
  - Replacing all properties of a node or relationship using the **=** operator
  - Mutating specific properties of a node or relationship 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
  - Equality and comparison of values
  - Ordering and comparison of values
  - Chaining comparison operations
  - Using a regular expression with **=~** to filter words
- Boolean operators
- Using boolean operators to filter numbers
- **String operators**
  - Concatenating two strings using +
- **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
- **Map operators**
  - Statically accessing the value of a nested map by key using the . operator
  - Dynamically accessing the value of a map by key using the [\ operator and a parameter]
- **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]
  - Dynamically accessing an element in a list using the [\ operator and a parameter]
  - Using IN with [\ on a nested list]

### 2.7.1. Operators at a glance

<table>
<thead>
<tr>
<th>Aggregation operators</th>
<th>DISTINCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property operators</td>
<td>. for static property access, [] for dynamic property access, = for replacing all properties, += for mutating specific properties</td>
</tr>
<tr>
<td>Mathematical operators</td>
<td>+, - , *, / , % , ^</td>
</tr>
<tr>
<td>Comparison operators</td>
<td>= , &lt; , &gt; , &lt;= , &gt;= , IS NULL , IS NOT NULL</td>
</tr>
<tr>
<td>String-specific comparison operators</td>
<td>STARTS WITH , ENDS WITH , CONTAINS , = for regex matching</td>
</tr>
<tr>
<td>Boolean operators</td>
<td>AND , OR , XOR , NOT</td>
</tr>
<tr>
<td>String operators</td>
<td>+ for concatenation</td>
</tr>
<tr>
<td>Temporal operators</td>
<td>+ and - for operations between durations and temporal instants/durations , * and / for operations between durations and numbers</td>
</tr>
<tr>
<td>Map operators</td>
<td>. for static value access by key , [] for dynamic value access by key</td>
</tr>
<tr>
<td>List operators</td>
<td>+ for concatenation , IN to check existence of an element in a list , [] for accessing element(s) dynamically</td>
</tr>
</tbody>
</table>
2.7.2. Aggregation operators

The aggregation operators comprise:

- remove duplicates values: `DISTINCT`

Using the `DISTINCT` operator

Retrieve the unique eye colors from `Person` nodes.

Query

```graphql
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

<table>
<thead>
<tr>
<th>p.eyeColor</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;blue&quot;</td>
</tr>
<tr>
<td>&quot;brown&quot;</td>
</tr>
</tbody>
</table>

Rows: 2  
Nodes created: 3  
Properties set: 6  
Labels added: 3

`DISTINCT` is commonly used in conjunction with aggregating functions.

2.7.3. Property operators

The property operators pertain to a node or a relationship, and comprise:

- statically access the property of a node or relationship using the dot operator: .
- dynamically access the property of a node or relationship using the subscript operator: []
- property replacement = for replacing all properties of a node or relationship
- property mutation operator += for setting specific properties of a node or relationship

Statically accessing a property of a node or relationship using the . operator
Query

```
CREATE
  (a:Person {name: 'Jane', livesIn: 'London'}),
  (b:Person {name: 'Tom', livesIn: 'Copenhagen'})
WITH a, b
MATCH (p:Person)
RETURN p.name
```

Table 7. Result

<table>
<thead>
<tr>
<th>p.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Jane&quot;</td>
</tr>
<tr>
<td>&quot;Tom&quot;</td>
</tr>
</tbody>
</table>

Rows: 2
Nodes created: 2
Properties set: 4
Labels added: 2

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

<table>
<thead>
<tr>
<th>restaurant.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Hungry Jo&quot;</td>
</tr>
</tbody>
</table>

Rows: 1
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.

Replacing all properties of a node or relationship using the `=` operator

Query

```
CREATE (a:Person {name: 'Jane', age: 20})
WITH a
MATCH (p:Person {name: 'Jane'})
SET p = {name: 'Ellen', livesIn: 'London'}
RETURN p.name, p.age, p.livesIn
```
All the existing properties on the node are replaced by those provided in the map; i.e. the name property is updated from Jane to Ellen, the age property is deleted, and the livesIn property is added.

Table 9. Result

<table>
<thead>
<tr>
<th>p.name</th>
<th>p.age</th>
<th>p.livesIn</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Ellen&quot;</td>
<td>&lt;null&gt;</td>
<td>&quot;London&quot;</td>
</tr>
</tbody>
</table>

Rows: 1  
Nodes created: 1  
Properties set: 5  
Labels added: 1

See **Replace all properties using a map and \(=\)** for more details on using the property replacement operator \(=\).

Mutating specific properties of a node or relationship using the \(+=\) operator

**Query**

```
CREATE (a:Person {name: 'Jane', age: 20})
WITH a
MATCH (p:Person {name: 'Jane'})
SET p += {name: 'Ellen', livesIn: 'London'}
RETURN p.name, p.age, p.livesIn
```

The properties on the node are updated as follows by those provided in the map: the name property is updated from Jane to Ellen, the age property is left untouched, and the livesIn property is added.

Table 10. Result

<table>
<thead>
<tr>
<th>p.name</th>
<th>p.age</th>
<th>p.livesIn</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Ellen&quot;</td>
<td>20</td>
<td>&quot;London&quot;</td>
</tr>
</tbody>
</table>

Rows: 1  
Nodes created: 1  
Properties set: 4  
Labels added: 1

See **Mutate specific properties using a map and \(+=\)** for more details on using the property mutation operator \(+=\).

### 2.7.4. 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 11. Result

<table>
<thead>
<tr>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.0</td>
</tr>
</tbody>
</table>

Rows: 1

Using the unary minus operator -

Query

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

Table 12. Result

<table>
<thead>
<tr>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

Rows: 1

2.7.5. 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
Comparing two numbers

Query

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

Table 13. Result

<table>
<thead>
<tr>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
</tr>
</tbody>
</table>

Rows: 1

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

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

Table 14. Result

<table>
<thead>
<tr>
<th>candidate</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;John&quot;</td>
</tr>
<tr>
<td>&quot;Jonathan&quot;</td>
</tr>
</tbody>
</table>

Rows: 2

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

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 evaluates to 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. v IS NOT NULL is equivalent to NOT(v IS NULL).

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

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

Ordering and comparison of values

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

- Numerical values are compared for ordering using numerical order (e.g. 3 < 4 is true).
- All comparability tests (<, <=, >, >=) with java.lang.Double.NaN evaluate as false. For example, 1 > b and 1 < b are both false when b is NaN.
- 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 if 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. 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.

- 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.

- 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].

- Comparing for ordering when one argument is null (e.g. null < 3 is null).

- Ordering of values with different types:
  - The ordering is, in ascending order, defined according to the following list:
    - Map
    - Node
    - Relationship
    - List
    - Path
    - DateTime
    - LocalDateTime
    - Date
    - Time
    - LocalTime
    - Duration
The value `null` is considered larger than any value.

- **Ordering** of composite type values:
  - For the *composite types* (e.g. maps and lists), elements of the containers are compared pairwise for ordering and thus determine the ordering of two container types. For example, `[1, 'foo', 3]` is ordered before `[1, 2, 'bar']` since `'foo'` is ordered before 2.

### 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:

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

is equivalent to

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

Thus, it matches all nodes where the age is between 21 and 30.

This syntax extends to all equality `=` and inequality `<>` comparisons, as well as to chains longer than three.

Chains of `=` and `<>` are treated in a special way in Cypher.

This means that `1=1=true` is equivalent to `1=1 AND 1=true` and not to `(1=1)=true` or `1=(1=true)`.

For example:

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

Is equivalent to:

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

[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 divisible by 400. This means that over 400 years there are ((365 * 4 + 1)
\[ (25 - 1) \times 4 + 1 = 146097 \] days, which means an average of 365.2425 days per year.
Chapter 3. Using a regular expression with =~ to filter words

Query

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

Table 15. Result

<table>
<thead>
<tr>
<th>word</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;mouse&quot;</td>
</tr>
<tr>
<td>&quot;house&quot;</td>
</tr>
</tbody>
</table>

Rows: 2

Further information and examples regarding the use of regular expressions in filtering can be found in Regular expressions.
Chapter 4. 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**.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>a AND b</th>
<th>a OR b</th>
<th>a XOR b</th>
<th>NOT a</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>null</td>
<td>false</td>
<td>null</td>
<td>null</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>null</td>
<td>false</td>
<td>false</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>null</td>
<td>true</td>
<td>null</td>
<td>true</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

4.1. Using boolean operators to filter numbers

Query

```sql
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 16. Result

<table>
<thead>
<tr>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>

Rows: 3
Chapter 5. String operators

The string operators comprise:

- concatenating strings: +

5.1. Concatenating two strings with +

Query

```
RETURN 'neo' + '4j' AS result
```

Table 17. Result

<table>
<thead>
<tr>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;neo4j&quot;</td>
</tr>
</tbody>
</table>

Rows: 1
Chapter 6. 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:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Left-hand operand</th>
<th>Right-hand operand</th>
<th>Type of result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Temporal instant</td>
<td>Duration</td>
<td>The type of the temporal instant</td>
</tr>
<tr>
<td>+</td>
<td>Duration</td>
<td>Temporal instant</td>
<td>The type of the temporal instant</td>
</tr>
<tr>
<td>-</td>
<td>Temporal instant</td>
<td>Duration</td>
<td>The type of the temporal instant</td>
</tr>
<tr>
<td>+</td>
<td>Duration</td>
<td>Duration</td>
<td>Duration</td>
</tr>
<tr>
<td>-</td>
<td>Duration</td>
<td>Duration</td>
<td>Duration</td>
</tr>
<tr>
<td>*</td>
<td>Duration</td>
<td>Number</td>
<td>Duration</td>
</tr>
<tr>
<td>*</td>
<td>Number</td>
<td>Duration</td>
<td>Duration</td>
</tr>
<tr>
<td>/</td>
<td>Duration</td>
<td>Number</td>
<td>Duration</td>
</tr>
</tbody>
</table>

6.1. Adding and subtracting a Duration to or from a temporal instant

Query

```sql
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 18. Result

<table>
<thead>
<tr>
<th>aDateTime + aDuration</th>
<th>aDateTime - aDuration</th>
</tr>
</thead>
</table>

Rows: 1

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**

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

**Table 19. Result**

<table>
<thead>
<tr>
<th>aDate + aDuration</th>
<th>aDate - aDuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1996-10-11</td>
<td>1972-10-11</td>
</tr>
</tbody>
</table>

Rows: 1

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**

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

**Table 20. Result**

<table>
<thead>
<tr>
<th>date1</th>
<th>date2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2012-02-28</td>
<td>2012-02-29</td>
</tr>
</tbody>
</table>

Rows: 1

### 6.2. Adding and subtracting a Duration to or from another Duration

**Query**

```sql
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 21. Result**

<table>
<thead>
<tr>
<th>duration1</th>
<th>duration2</th>
<th>duration1 + duration2</th>
<th>duration1 - duration2</th>
</tr>
</thead>
<tbody>
<tr>
<td>P12Y5M14DT16H13M10.0000000001S</td>
<td>P1M-14DT1SH49M10S</td>
<td>P12Y6MT32H2M20.0000000001S</td>
<td>P12Y4M28DT24M0.0000000001S</td>
</tr>
</tbody>
</table>

Rows: 1
6.3. 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

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

Table 22. Result

<table>
<thead>
<tr>
<th>aDuration</th>
<th>aDuration * 2</th>
<th>aDuration / 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P14DT13M10.00000001S</td>
<td>P28DT26M20.00000002S</td>
<td>P4DT16H4M23.333333333S</td>
</tr>
</tbody>
</table>

Rows: 1
Chapter 7. Map operators

The map operators comprise:

- statically access the value of a map by key using the dot operator: `.`
- dynamically access the value of a map by key using the subscript operator: `[]`

The behavior of the `[]` operator with respect to `null` is detailed in The `[]` operator and `null`.

7.1. Statically accessing the value of a nested map by key using the `. ` operator

Query

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

Table 23. Result

<table>
<thead>
<tr>
<th>p.person.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Anne&quot;</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

7.2. Dynamically accessing the value of a map by key using the `[]` operator and a parameter

A parameter may be used to specify the key of the value to access:

Parameters

```
{
   "myKey" : "name"
}
```

Query

```
WITH {name: 'Anne', age: 25} AS a
RETURN a["myKey"] AS result
```

Table 24. Result

<table>
<thead>
<tr>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Anne&quot;</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

More details on maps can be found in Maps.
Chapter 8. List operators

The list operators comprise:

- concatenating lists $l_1$ and $l_2$: $[l_1] + [l_2]$
- checking if an element $e$ exists in a list $l$: $e \text{ IN } [l]$
- dynamically 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.

8.1. Concatenating two lists using +

Query

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

Table 25. Result

<table>
<thead>
<tr>
<th>myList</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,2,3,4,5,6,7]</td>
</tr>
</tbody>
</table>

Rows: 1

8.2. 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 \text{ IN } [2, 3, 8]
RETURN number
```

Table 26. Result

<table>
<thead>
<tr>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

Rows: 2

8.3. 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 innerList are compared pairwise in ascending order from the first element in innerList to the last element in innerList.
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 27. Result

<table>
<thead>
<tr>
<th>inList</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

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 28. Result

<table>
<thead>
<tr>
<th>inList</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

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

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

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

### 8.4. Accessing elements in a list using the \([\ldots]\) 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 29. Result**

<table>
<thead>
<tr>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>[&quot;John&quot;,&quot;Bill&quot;]</td>
</tr>
</tbody>
</table>

Rows: 1

8.5. Dynamically accessing an element in a list using the [] operator and a parameter

A parameter may be used to specify the index of the element to access:

**Parameters**

```json
{
   "myIndex" : 1
}
```

**Query**

```sql
WITH ['Anne', 'John', 'Bill', 'Diane', 'Eve'] AS names
RETURN names[$myIndex] AS result
```

**Table 30. Result**

<table>
<thead>
<tr>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;John&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

8.6. Using IN with [] on a nested list

**IN** can be used in conjunction with [] to test whether an element exists in a nested list:

**Parameters**

```json
{
   "myIndex" : 1
}
```

**Query**

```sql
WITH [[1, 2, 3]] AS l
RETURN 3 IN l[0] AS result
```

**Table 31. Result**

<table>
<thead>
<tr>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
</tr>
</tbody>
</table>
More details on lists can be found in Lists in general.

8.7. Comments

This section describes how to use comments in Cypher.

A comment begins with double slash (//) and continues to the end of the line. Comments do not execute, they are for humans to read.

Examples:

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

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

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

8.8. Patterns

This section contains an overview of data patterns in Cypher.

- Introduction
- Patterns for nodes
- Patterns for related nodes
- Patterns for labels
- Specifying properties
- Patterns for relationships
- Variable-length pattern matching
- Assigning to path variables

8.8.1. 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 are 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

### 8.8.2. 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`.

### 8.8.3. 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)
```
8.8.4. 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:

\[(a:User)\rightarrow(b)\]

One can also describe a node that has multiple labels:

\[(a:User:Admin)\rightarrow(b)\]

8.8.5. 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:

\[(a\{name: 'Andy', sport: 'Brazilian Ju-Jitsu'\})\]

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.

8.8.6. 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:

\[(a)\rightarrow(b)\]

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)

### 8.8.7. 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 Cypher path matching. 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 range of lengths can also be specified: such relationship patterns are called 'variable length relationships'. For example:

\[(a)\rightarrow(*)\rightarrow(b)\]

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:

\[(a)\rightarrow(*3..)\rightarrow(b)\]

To describe paths of length 5 or less, use:

\[(a)\rightarrow(*..5)\rightarrow(b)\]

Omitting both bounds is equivalent to specifying a minimum of 1, allowing paths of any positive length to be described:

\[(a)\rightarrow(*)\rightarrow(b)\]

As a simple example, let's take the graph and query below:

Graph

Query

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

Table 32. Result

<table>
<thead>
<tr>
<th>remote_friend.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Dilshad&quot;</td>
</tr>
</tbody>
</table>
This query finds data in the graph with a shape that fits the pattern: specifically a node (with the name property 'Filipa') and then the `KNOWS` related nodes, one or two hops away. This is a typical example of finding first and second degree friends.

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

8.8.8. 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.

8.9. 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.

- Refer to Temporal functions - instant types 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.

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Date support</th>
<th>Time support</th>
<th>Time zone support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LocalTime</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>DateTime</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LocalDateTime</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Duration</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

`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 captures the amount of time between two instants, it does not capture a start time and end time.

8.9.1. 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). 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).
- Specifying a named time zone.
- Specifying both the offset and the time zone name (with the requirement that these match).

See Specifying time zones for examples.

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.

8.9.2. Temporal instants
Specifying temporal instants

A temporal instant consists of three parts; the date, the time, and the timezone. These parts can be combined to produce the various temporal value types. The character T is a literal character.

<table>
<thead>
<tr>
<th>Temporal instant type</th>
<th>Composition of parts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>&lt;date&gt;</td>
</tr>
<tr>
<td>Time</td>
<td>&lt;time&gt;&lt;timezone&gt; or T&lt;time&gt;&lt;timezone&gt;</td>
</tr>
<tr>
<td>LocalTime</td>
<td>&lt;time&gt; or T&lt;time&gt;</td>
</tr>
<tr>
<td>DateTime*</td>
<td>&lt;date&gt;T&lt;time&gt;&lt;timezone&gt;</td>
</tr>
<tr>
<td>LocalDateTime*</td>
<td>&lt;date&gt;T&lt;time&gt;</td>
</tr>
</tbody>
</table>

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

Specifying dates

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

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

- Minus sign, - must prefix any year before 0000, (e.g. -3000-01-01).
- Plus sign, + must prefix any year after 9999, (e.g. +11000-01-01).
- The year must be separated with - from the next component:
  - if the next component is month, (e.g. +11000-01).
If the next component is day of the year, (e.g. +11000-123).

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:

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

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

<table>
<thead>
<tr>
<th>Component</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hour</td>
<td>HH</td>
<td>Specified with a double digit number from 00 to 23.</td>
</tr>
<tr>
<td>Minute</td>
<td>MM</td>
<td>Specified with a double digit number from 00 to 59.</td>
</tr>
<tr>
<td>Component</td>
<td>Format</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>Second</td>
<td>SS</td>
<td>Specified with a double digit number from 00 to 59.</td>
</tr>
<tr>
<td>fraction</td>
<td>sssssss</td>
<td>Specified with a number from 0 to 999999999. It is not required to specify trailing zeros. fraction is an optional, sub-second component of Second. This can be separated from Second using either a full stop (.) or a comma (,). The fraction is in addition to the two digits of Second.</td>
</tr>
</tbody>
</table>

Cypher does not support leap seconds; 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:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
<th>Interpretation of example</th>
</tr>
</thead>
<tbody>
<tr>
<td>HHMMSS.sssssssss</td>
<td>Hour:Minute:Second.fraction</td>
<td>214032.142</td>
<td>21:40:32.142</td>
</tr>
<tr>
<td>HHMMSS</td>
<td>Hour:Minute:Second</td>
<td>214032</td>
<td>21:40:32.000</td>
</tr>
<tr>
<td>HH:MM</td>
<td>Hour:Minute</td>
<td>21:40</td>
<td>21:40:00.000</td>
</tr>
<tr>
<td>HHM</td>
<td>Hour:Minute</td>
<td>2140</td>
<td>21:40:00.000</td>
</tr>
<tr>
<td>HH</td>
<td>Hour</td>
<td>21</td>
<td>21:00:00.000</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
<th>Supported for DateTime</th>
<th>Supported for Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>UTC</td>
<td>Z</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>±HH:MM</td>
<td>Hour:Minute</td>
<td>+09:30</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>±HH:MM[ZoneName]</td>
<td>Hour:Minute[ZoneName]</td>
<td>+08:45[Australia/Eucla]</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>±HHMM</td>
<td>Hour:Minute</td>
<td>+0100</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>±HHMM[ZoneName]</td>
<td>Hour:Minute[ZoneName]</td>
<td>+0200[Africa/Johannesburg]</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>±HH</td>
<td>Hour</td>
<td>-08</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>±HH[ZoneName]</td>
<td>Hour[ZoneName]</td>
<td>+08[Asia/Singapore]</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>[ZoneName]</td>
<td>[ZoneName]</td>
<td>[America/Regina]</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

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 33. Result

<table>
<thead>
<tr>
<th>theDateTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015-06-24T12:50:35.556+01:00</td>
</tr>
</tbody>
</table>

Rows: 1

Parsing a LocalDateTime using the ordinal date format:

Query

```
RETURN localdatetime('2015185T19:32:24') AS theLocalDateTime
```
Table 34. Result

<table>
<thead>
<tr>
<th>theLocalDateTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015-07-04T19:32:24</td>
</tr>
</tbody>
</table>

Rows: 1

Parsing a Date using the week date format:

Query

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

Table 35. Result

<table>
<thead>
<tr>
<th>theDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015-03-26</td>
</tr>
</tbody>
</table>

Rows: 1

Parsing a Time:

Query

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

Table 36. Result

<table>
<thead>
<tr>
<th>theTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:50:35.556+01:00</td>
</tr>
</tbody>
</table>

Rows: 1

Parsing a LocalTime:

Query

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

Table 37. Result

<table>
<thead>
<tr>
<th>theLocalTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:50:35.556</td>
</tr>
</tbody>
</table>

Rows: 1

Accessing components of temporal instants

Components of temporal instant values can be accessed as properties.

Table 38. Components of temporal instant values and where they are supported
<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
<th>Type</th>
<th>Range/Format</th>
<th>Date</th>
<th>DateTime</th>
<th>LocalDateTime</th>
<th>Time</th>
<th>LocalTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>instant.year</td>
<td>The year component represents the astronomical year number of the instant.[2]</td>
<td>Integer</td>
<td>At least 4 digits. For more information, see the rules for using the Year component</td>
<td>✔</td>
<td>✔</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>instant.quarter</td>
<td>The quarter-of-the-year component.</td>
<td>Integer</td>
<td>1 to 4</td>
<td>✔</td>
<td>✔</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>instant.month</td>
<td>The month-of-the-year component.</td>
<td>Integer</td>
<td>1 to 12</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>instant.week</td>
<td>The week-of-the-year component.</td>
<td>Integer</td>
<td>1 to 53</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>instant.weekYear</td>
<td>The year that the week-of-year component belongs to.[3]</td>
<td>Integer</td>
<td>At least 4 digits. For more information, see the rules for using the Year component</td>
<td>✔</td>
<td>✔</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>instant.dayOfQuarter</td>
<td>The day-of-the-quarter component.</td>
<td>Integer</td>
<td>1 to 92</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>instant.quarterDay</td>
<td>The day-of-the-quarter component. (alias for instant.dayOfQuarter)</td>
<td>Integer</td>
<td>1 to 92</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>instant.day</td>
<td>The day-of-the-month component.</td>
<td>Integer</td>
<td>1 to 31</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>instant.ordinalDay</td>
<td>The day-of-the-year component.</td>
<td>Integer</td>
<td>1 to 366</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>instant.dayOfWeek</td>
<td>The day-of-the-week component (the first day of the week is Monday).</td>
<td>Integer</td>
<td>1 to 7</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>instant.weekDay</td>
<td>The day-of-the-week component (alias for instant.dayOfWeek).</td>
<td>Integer</td>
<td>1 to 7</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>instant.hour</td>
<td>The hour component.</td>
<td>Integer</td>
<td>0 to 23</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>instant.minute</td>
<td>The minute component.</td>
<td>Integer</td>
<td>0 to 59</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Component</td>
<td>Description</td>
<td>Type</td>
<td>Range/Format</td>
<td>Date</td>
<td>DateTime</td>
<td>LocalDate</td>
<td>Time</td>
<td>LocalTime</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------------------------------</td>
<td>-----------</td>
<td>--------------</td>
<td>------</td>
<td>----------</td>
<td>------------</td>
<td>------</td>
<td>-----------</td>
</tr>
<tr>
<td>instant.second</td>
<td>The second component.</td>
<td>Integer</td>
<td>0 to 59</td>
<td>☑</td>
<td>☑</td>
<td></td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>instant.millisecond</td>
<td>The millisecond component.</td>
<td>Integer</td>
<td>0 to 999</td>
<td>☑</td>
<td>☑</td>
<td></td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>instant.microsecond</td>
<td>The microsecond component.</td>
<td>Integer</td>
<td>0 to 999999</td>
<td>☑</td>
<td>☑</td>
<td></td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>instant.nanosecond</td>
<td>The nanosecond component.</td>
<td>Integer</td>
<td>0 to 9999999</td>
<td>☑</td>
<td>☑</td>
<td></td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>instant.timezone</td>
<td>The timezone component.</td>
<td>String</td>
<td>Depending on how the time zone was specified, this is either a time zone name or an offset from UTC in the format ±HHMM</td>
<td>☑</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instant.offset</td>
<td>The timezone offset</td>
<td>String</td>
<td>±HHMM</td>
<td>☑</td>
<td>☑</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instant.offsetMinutes</td>
<td>The timezone offset in minutes</td>
<td>Integer</td>
<td>-1080 to +1080</td>
<td>☑</td>
<td>☑</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instant.offsetSeconds</td>
<td>The timezone offset in seconds</td>
<td>Integer</td>
<td>-64800 to +64800</td>
<td>☑</td>
<td>☑</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instant.epochMilliseconds</td>
<td>The number of milliseconds between 1970-01-01T00:00:00+0000 and the instant.</td>
<td>Integer</td>
<td>Positive for instants after and negative for instants before 1970-01-01T00:00:00+0000</td>
<td>☑</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instant.epochSeconds</td>
<td>The number of seconds between 1970-01-01T00:00:00+0000 and the instant.</td>
<td>Integer</td>
<td>Positive for instants after and negative for instants before 1970-01-01T00:00:00+0000</td>
<td>☑</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 39. Result

<table>
<thead>
<tr>
<th>d.year</th>
<th>d.quarter</th>
<th>d.month</th>
<th>d.week</th>
<th>d.weekYear</th>
<th>d.day</th>
<th>d.ordinalDay</th>
<th>d.dayOfWeek</th>
<th>d.dayOfQuarter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984</td>
<td>10</td>
<td>10</td>
<td>41</td>
<td>1984</td>
<td>11</td>
<td>285</td>
<td>4</td>
<td>11</td>
</tr>
</tbody>
</table>
The following query shows how to extract the date related components of a DateTime value:

Query

```sql
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
```

Table 40. Result

<table>
<thead>
<tr>
<th>d.year</th>
<th>d.quarter</th>
<th>d.month</th>
<th>d.week</th>
<th>d.weekYear</th>
<th>d.day</th>
<th>d.ordinalDay</th>
<th>d.dayOfWeek</th>
<th>d.dayOfQuarter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984</td>
<td>4</td>
<td>11</td>
<td>45</td>
<td>1984</td>
<td>11</td>
<td>316</td>
<td>7</td>
<td>42</td>
</tr>
</tbody>
</table>

Rows: 1

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

Query

```sql
WITH datetime({
    year: 1984, month: 11, day: 11,
    hour: 12, minute: 31, second: 14, nanosecond: 645876123,
    timezone: 'Europe/Stockholm'
}) AS d
RETURN d.hour, d.minute, d.second, d.millisecond, d.microsecond, d.nanosecond
```

Table 41. Result

<table>
<thead>
<tr>
<th>d.hour</th>
<th>d.minute</th>
<th>d.second</th>
<th>d.millisecond</th>
<th>d.microsecond</th>
<th>d.nanosecond</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>31</td>
<td>14</td>
<td>645</td>
<td>645876</td>
<td>645876123</td>
</tr>
</tbody>
</table>

Rows: 1

The following query shows how to extract the epoch time and timezone related components of a DateTime value:

Query

```sql
WITH datetime({
    year: 1984, month: 11, day: 11,
    hour: 12, minute: 31, second: 14, nanosecond: 645876123,
    timezone: 'Europe/Stockholm'
}) AS d
RETURN d.timezone, d.offset, d.offsetMinutes, d.epochSeconds, d.epochMillis
```

Table 42. Result
8.9.3. 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 within the bounds of a 64-bit integer.
  - 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.
  - The maximum total length of a Duration is bounded by the number of seconds that can be held in a 64-bit integer.

- **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:

<table>
<thead>
<tr>
<th>Component identifier</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>Years</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>Months</td>
<td>Must be specified before ( T ).</td>
</tr>
<tr>
<td>W</td>
<td>Weeks</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>Days</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>Hours</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>Minutes</td>
<td>Must be specified after ( T ).</td>
</tr>
<tr>
<td>S</td>
<td>Seconds</td>
<td></td>
</tr>
</tbody>
</table>
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 43. Result

<table>
<thead>
<tr>
<th>theDuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>P14DT16H12M</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

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

Query

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

Table 44. Result

<table>
<thead>
<tr>
<th>theDuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>P5M1DT12H</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

Return a Duration of 45 seconds:

Query

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

Table 45. Result

<table>
<thead>
<tr>
<th>theDuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>PT45S</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

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

Query

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

Table 46. Result
Accessing components of durations

A `Duration` can have several components, each categorized into Months, Days, and Seconds groups.

Components of `Duration` values are truncated within their component groups as follows:

<table>
<thead>
<tr>
<th>Component Group</th>
<th>Component</th>
<th>Description</th>
<th>Type</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Months</td>
<td>duration.years</td>
<td>The total number of years</td>
<td>Integer</td>
<td>Each set of 4 quarters is counted as 1 year; each set of 12 months is counted as 1 year.</td>
</tr>
<tr>
<td></td>
<td>duration.quarters</td>
<td>The total number of quarters</td>
<td>Integer</td>
<td>Each year is counted as 4 quarters; each set of 3 months is counted as 1 quarter.</td>
</tr>
<tr>
<td></td>
<td>duration.months</td>
<td>The total number of months</td>
<td>Integer</td>
<td>Each year is counted as 12 months; each quarter is counted as 3 months.</td>
</tr>
<tr>
<td>Days</td>
<td>duration.weeks</td>
<td>The total number of weeks</td>
<td>Integer</td>
<td>Each set of 7 days is counted as 1 week.</td>
</tr>
<tr>
<td></td>
<td>duration.days</td>
<td>The total number of days</td>
<td>Integer</td>
<td>Each week is counted as 7 days.</td>
</tr>
<tr>
<td>Seconds</td>
<td>duration.hours</td>
<td>The total number of hours</td>
<td>Integer</td>
<td>Each set of 60 minutes is counted as 1 hour; each set of 3600 seconds is counted as 1 hour.</td>
</tr>
<tr>
<td></td>
<td>duration.minutes</td>
<td>The total number of minutes</td>
<td>Integer</td>
<td>Each hour is counted as 60 minutes; each set of 60 seconds is counted as 1 minute.</td>
</tr>
<tr>
<td></td>
<td>duration.seconds</td>
<td>The total number of seconds</td>
<td>Integer</td>
<td>Each hour is counted as 3600 seconds; each minute is counted as 60 seconds.</td>
</tr>
<tr>
<td></td>
<td>duration.milliseconds</td>
<td>The total number of milliseconds</td>
<td>Integer</td>
<td>Each set of 1000 milliseconds is counted as 1 second.</td>
</tr>
<tr>
<td></td>
<td>duration.microseconds</td>
<td>The total number of microseconds</td>
<td>Integer</td>
<td>Each millisecond is counted as 1000 microseconds.</td>
</tr>
<tr>
<td></td>
<td>duration.nanoseconds</td>
<td>The total number of nanoseconds</td>
<td>Integer</td>
<td>Each microsecond is counted as 1000 nanoseconds.</td>
</tr>
</tbody>
</table>
Please note that:

- Cypher uses UTC-SLS when handling leap seconds.
- 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.

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:

<table>
<thead>
<tr>
<th>Component</th>
<th>Component Group</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>duration.quartersOfYear</td>
<td>Months</td>
<td>The number of quarters in the group that do not make a whole year</td>
<td>Integer</td>
</tr>
<tr>
<td>duration.monthsOfYear</td>
<td>Months</td>
<td>The number of months in the group that do not make a whole year</td>
<td>Integer</td>
</tr>
<tr>
<td>duration.monthsOfQuarter</td>
<td>Months</td>
<td>The number of months in the group that do not make a whole quarter</td>
<td>Integer</td>
</tr>
<tr>
<td>duration.daysOfWeek</td>
<td>Days</td>
<td>The number of days in the group that do not make a whole week</td>
<td>Integer</td>
</tr>
<tr>
<td>duration.minutesOfHour</td>
<td>Seconds</td>
<td>The number of minutes in the group that do not make a whole hour</td>
<td>Integer</td>
</tr>
<tr>
<td>duration.secondsOfMinute</td>
<td>Seconds</td>
<td>The number of seconds in the group that do not make a whole minute</td>
<td>Integer</td>
</tr>
<tr>
<td>duration.millisecondsOfSecond</td>
<td>Seconds</td>
<td>The number of milliseconds in the group that do not make a whole second</td>
<td>Integer</td>
</tr>
<tr>
<td>duration.microsecondsOfSecond</td>
<td>Seconds</td>
<td>The number of microseconds in the group that do not make a whole second</td>
<td>Integer</td>
</tr>
<tr>
<td>duration.nanosecondsOfSecond</td>
<td>Seconds</td>
<td>The number of nanoseconds in the group that do not make a whole second</td>
<td>Integer</td>
</tr>
</tbody>
</table>

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

**Query**

```sql
WITH duration({years: 1, months: 5, days: 111, minutes: 42}) AS d
RETURN d.years, d.quarters, d.quartersOfYear, d.months, d.monthsOfYear, d.monthsOfQuarter
```

**Table 47. Result**

<table>
<thead>
<tr>
<th>d.years</th>
<th>d.quarters</th>
<th>d.quartersOfYear</th>
<th>d.months</th>
<th>d.monthsOfYear</th>
<th>d.monthsOfQuarter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>1</td>
<td>17</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

Rows: 1
The following query shows how to extract the day based components of a Duration value:

**Query**

```sql
WITH duration({
  months: 5,
  days: 25,
  hours: 1
}) AS d
RETURN d.weeks, d.days, d.daysOfWeek
```

**Table 48. Result**

<table>
<thead>
<tr>
<th>d.weeks</th>
<th>d.days</th>
<th>d.daysOfWeek</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>25</td>
<td>4</td>
</tr>
</tbody>
</table>

Rows: 1

The following query shows how to extract the most significant second based components of a Duration value:

**Query**

```sql
WITH duration({
  years: 1,
  months: 1,
  days: 1,
  hours: 1,
  minutes: 1,
  seconds: 1,
  nanoseconds: 111111111
}) AS d
RETURN d.hours, d.minutes, d.seconds, d.milliseconds, d.microseconds, d.nanoseconds
```

**Table 49. Result**

<table>
<thead>
<tr>
<th>d.hours</th>
<th>d.minutes</th>
<th>d.seconds</th>
<th>d.milliseconds</th>
<th>d.microseconds</th>
<th>d.nanoseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>61</td>
<td>3661</td>
<td>3661111</td>
<td>3661111111</td>
<td>3661111111111</td>
</tr>
</tbody>
</table>

Rows: 1

The following query shows how to extract the less significant second based components of a Duration value:

**Query**

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

**Table 50. Result**

<table>
<thead>
<tr>
<th>d.minutesOfHour</th>
<th>d.secondsOfMinute</th>
<th>d.millisecondsOfSecond</th>
<th>d.microsecondsOfSecond</th>
<th>d.nanosecondsOfSecond</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>111</td>
<td>111111</td>
<td>111111111</td>
</tr>
</tbody>
</table>

Rows: 1

### 8.9.4. Examples

The following examples illustrate the use of some of the temporal functions and operators. Refer to
Temporal functions - instant types and Temporal operators for more details.

Create a Duration representing 1.5 days:

Query

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

Table 51. Result

<table>
<thead>
<tr>
<th>theDuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1DT12H</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

Compute the Duration between two temporal instants:

Query

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

Table 52. Result

<table>
<thead>
<tr>
<th>theDuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>P30Y8M13D</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

Compute the number of days between two Date values:

Query

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

Table 53. Result

<table>
<thead>
<tr>
<th>theDuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>P299D</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

Get the first Date of the current year:

Query

```
RETURN date.truncate('year') AS day
```

Table 54. Result

<table>
<thead>
<tr>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>2022-01-01</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>
Get the Date of the Thursday in the week of a specific date:

Query

```sql
RETURN date.truncate('week', date('2019-10-01'), {dayOfWeek: 4}) AS thursday
```

Table 55. Result

<table>
<thead>
<tr>
<th>thursday</th>
</tr>
</thead>
<tbody>
<tr>
<td>2019-10-03</td>
</tr>
</tbody>
</table>

Rows: 1

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

Query

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

Table 56. Result

<table>
<thead>
<tr>
<th>lastDay</th>
</tr>
</thead>
<tbody>
<tr>
<td>2022-09-30</td>
</tr>
</tbody>
</table>

Rows: 1

Add a Duration to a Date:

Query

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

Table 57. Result

<table>
<thead>
<tr>
<th>theTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>01:42:19Z</td>
</tr>
</tbody>
</table>

Rows: 1

Add two Duration values:

Query

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

Table 58. Result

<table>
<thead>
<tr>
<th>theDuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1M2DT25H</td>
</tr>
</tbody>
</table>

Rows: 1

Multiply a Duration by a number:
Query

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

Table 59. Result

<table>
<thead>
<tr>
<th>theDuration</th>
<th>PT74H54M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows:</td>
<td>1</td>
</tr>
</tbody>
</table>

Divide a Duration by a number:

Query

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

Table 60. Result

<table>
<thead>
<tr>
<th>theDuration</th>
<th>PT1H38M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows:</td>
<td>1</td>
</tr>
</tbody>
</table>

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 61. Result

<table>
<thead>
<tr>
<th>lessThanOneDayApart</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows:</td>
<td>1</td>
</tr>
</tbody>
</table>

Return the abbreviated name of the current month:

Query

```
```

Table 62. Result

<table>
<thead>
<tr>
<th>month</th>
<th>&quot;Aug&quot;</th>
</tr>
</thead>
</table>

75
8.9.5. 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.

8.10. 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
- Comparability and orderability

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.

8.10.1. 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 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).

### 8.10.2. 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 [Ordering and comparison of values](#).

#### Geographic coordinate reference systems

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

- **WGS 84 2D**
  - 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**
  - 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:

\[
\text{RETURN point.distance(a,b) / 1000 AS km}
\]

An example demonstrating conversion on incoming and outgoing values is:

Query

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

Table 63. Result

<table>
<thead>
<tr>
<th>km</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
</tr>
</tbody>
</table>

Rows: 1

Cartesian coordinate reference systems

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

- **Cartesian 2D**
  - 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**
  - 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
  p2d AS {
    x: 3,
    y: 0
  },
  p3d AS {
    x: 0,
    y: 4,
    z: 1
  }
RETURN
  point.distance(p2d, p3d) AS bad,
  point.distance(p2d, point({x: p3d.x, y: p3d.y})) AS good
```

Table 64. Result
8.10.3. 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 \texttt{latitude} and \texttt{longitude} the CRS will be assumed to be Geographic and therefore either \texttt{WGS-84} or \texttt{WGS-84-3D}.
  - If instead \texttt{x} and \texttt{y} are used, then the default CRS would be \texttt{Cartesian} or \texttt{Cartesian-3D}
- Number of dimensions:
  - If there are 2 dimensions in the coordinate, \texttt{x} & \texttt{y} or \texttt{longitude} & \texttt{latitude} the CRS will be a 2D CRS
  - If there is a third dimensions in the coordinate, \texttt{z} or \texttt{height} the CRS will be a 3D CRS

All fields are provided to the \texttt{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 \texttt{y} before \texttt{x} (\texttt{latitude} before \texttt{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 \texttt{point} function calls, and how those values are displayed in the results:

Query

\begin{verbatim}
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
\end{verbatim}

Table 65. Result

<table>
<thead>
<tr>
<th>cartesian_2d</th>
<th>cartesian_3d</th>
<th>geo_2d</th>
<th>geo_3d</th>
</tr>
</thead>
<tbody>
<tr>
<td>point({x: 3.0, y: 0.0, crs: 'cartesian'})</td>
<td>point({x: 0.0, y: 4.0, z: 1.0, crs: 'cartesian-3d'})</td>
<td>point({x: 56.0, y: 12.0, crs: 'wgs-84'})</td>
<td>point({x: 56.0, y: 12.0, z: 1000.0, crs: 'wgs-84-3d'})</td>
</tr>
</tbody>
</table>
For the geographic coordinates, it is important to note that the latitude value should always lie in the interval \([-90, 90]\) and any other value outside this range will throw an exception. The longitude value should always lie in the interval \([-180, 180]\) and any other value outside this range will be wrapped around to fit in this range. The height value and any cartesian coordinates are not explicitly restricted, and any value within the allowed range of the signed 64-bit floating point type will be accepted.

Accessing components of points

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

### Table 66. Components of point instances and where they are supported

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

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

**Query**

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

**Table 67. Result**

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>crs</th>
<th>srid</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.0</td>
<td>4.0</td>
<td>&quot;cartesian&quot;</td>
<td>7203</td>
</tr>
</tbody>
</table>

Rows: 1

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

**Query**

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

**Table 68. Result**
8.10.4. Spatial index

If there is a 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. If two range predicates are used, which define minimum and maximum points, this will effectively result in a bounding box query. In addition, queries using the distance function can, under the right conditions, also use the index, as described in the section 'Spatial distance searches'.

8.10.5. Comparability and orderability

The comparability and orderability of spacial values are due to change in an upcoming future release. This means that queries that rely on the comparison of two points using the inequality operators, <, <=, >, and >=, or the specific order of an ORDER BY n.point query will need to be rewritten.

The most efficient way to do this is to explicitly specify the ordering. For example, by using point.x, point.y in cartesian coordinates, or point.longitude and point.latitude in geographic coordinates.

8.11. 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.

8.11.1. 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 69. Result
In the examples, you 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, you can use the square brackets again. This extracts from the start index and up to, but not including, the end index.

<table>
<thead>
<tr>
<th>Query</th>
<th>Table 70. Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>RETURN range(0, 10)[3]</code></td>
<td>range(0, 10)[3]</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Query</th>
<th>Table 71. Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>RETURN range(0, 10)[-3]</code></td>
<td>range(0, 10)[-3]</td>
</tr>
<tr>
<td></td>
<td>8</td>
</tr>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Query</th>
<th>Table 72. Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>RETURN range(0, 10)[0..3]</code></td>
<td>range(0, 10)[0..3]</td>
</tr>
<tr>
<td>[0, 1, 2]</td>
<td></td>
</tr>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

| Query                          |                  |
| `RETURN range(0, 10)[0..-5]`   |                  |
Table 73. Result

<table>
<thead>
<tr>
<th>range(0, 10)[0..-5]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0, 1, 2, 3, 4, 5]</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

Query

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

Table 74. Result

<table>
<thead>
<tr>
<th>range(0, 10)[-5..]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[6, 7, 8, 9, 10]</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

Query

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

Table 75. Result

<table>
<thead>
<tr>
<th>range(0, 10)[..4]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0, 1, 2, 3]</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

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

Query

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

Table 76. Result

<table>
<thead>
<tr>
<th>range(0, 10)[15]</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

Query

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

Table 77. Result

<table>
<thead>
<tr>
<th>range(0, 10)[5..15]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[5, 6, 7, 8, 9, 10]</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>
You can get the size of a list as follows:

Query

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

Table 78. Result

<table>
<thead>
<tr>
<th>size(range(0, 10)[0..3])</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

### 8.11.2. 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 79. Result

<table>
<thead>
<tr>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0.0,8.0,64.0,216.0,512.0,1000.0]</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

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 80. Result

<table>
<thead>
<tr>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0,2,4,6,8,10]</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

Query

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

Table 81. Result

<table>
<thead>
<tr>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0.0,1.0,8.0,27.0,64.0,125.0,216.0,343.0,512.0,729.0,1000.0]</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>
8.11.3. Pattern comprehension

Pattern comprehension is a syntactic construct available in Cypher for creating a list based on matchings of a pattern. A pattern comprehension matches the specified pattern like a normal MATCH clause, with predicates like a normal WHERE clause, but yields a custom projection as specified.

The following graph is used for the pattern comprehension examples:

This example returns a list that contains the year when the movies was released. The pattern matching in the pattern comprehension looks for Matrix in the movie title and that the node a (Person node with the name Keanu Reeves) has a relationship with the movie.

Query

```
MATCH (a:Person {name: 'Keanu Reeves'})
RETURN [(a)-->(b:Movie) WHERE b.title CONTAINS 'Matrix' | b.released] AS years
```

Table 82. Result

<table>
<thead>
<tr>
<th>years</th>
</tr>
</thead>
</table>

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

This example returns a sorted list that contains years. The pattern matching in the pattern comprehension looks for movie nodes that has a relationship with the node a (Person node with the name Keanu Reeves).

Query

```
MATCH (a:Person {name: 'Keanu Reeves'})
WITH [(a)-->(b:Movie) | b.released] AS years
UNWIND years AS year
WITH year ORDER BY year
RETURN COLLECT(year) AS sorted_years
```

Table 83. Result

<table>
<thead>
<tr>
<th>sorted_years</th>
</tr>
</thead>
</table>

Rows: 1
8.12. Maps

This section describes how to use maps in Cyphers.

- Literal maps
- Map projection
  - Examples of map projection

The following graph is used for the examples below:

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.

8.12.1. Literal maps

Cypher supports construction of maps. The key names in a map must be of type String. If returned through an HTTP API call, a JSON object will be returned. If returned in Java, an object of type java.util.Map<String, Object> will be returned.

Query

```
RETURN {key: 'Value', listKey: [{inner: 'Map1'}, {inner: 'Map2'}]}
```

Table 84. Result

<table>
<thead>
<tr>
<th>key: 'Value', listKey: [{inner: 'Map1'}, {inner: 'Map2'}]</th>
</tr>
</thead>
<tbody>
<tr>
<td>{listKey -&gt; [{inner -&gt; &quot;Map1&quot;}, {inner -&gt; &quot;Map2&quot;}], key -&gt; &quot;Value&quot;}</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

8.12.2. 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.

The following conditions apply:

- If the `map_variable` points to a `null` value, the whole map projection will evaluate to `null`.

- The key names in a map must be of type `String`.

**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**

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

**Table 85. Result**

<table>
<thead>
<tr>
<th>actor</th>
</tr>
</thead>
</table>

Rows: 1

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**

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

**Table 86. Result**

<table>
<thead>
<tr>
<th>actor</th>
</tr>
</thead>
<tbody>
<tr>
<td>{nbrOfMovies -&gt; 2, name -&gt; &quot;Martin Sheen&quot;}</td>
</tr>
<tr>
<td>{nbrOfMovies -&gt; 3, name -&gt; &quot;Charlie Sheen&quot;}</td>
</tr>
</tbody>
</table>

Rows: 2
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**

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

**Table 87. Result**

<table>
<thead>
<tr>
<th>actor</th>
</tr>
</thead>
<tbody>
<tr>
<td>{realName -&gt; &quot;Carlos Irwin Estévez&quot;, name -&gt; &quot;Charlie Sheen&quot;, age -&gt; &lt;null&gt;}</td>
</tr>
</tbody>
</table>

Rows: 1

8.13. Working with **null**

This section describes working with the **null** value.

- Introduction to **null** in Cypher
- Logical operations with **null**
- The **IN** operator and **null**
- The **[\ operator and null**
- Expressions that return **null**
- **IS NULL and IS NOT NULL**

8.13.1. 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**.

8.13.2. 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**.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a AND b</th>
<th>a OR b</th>
<th>a XOR b</th>
<th>NOT a</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>
## 8.13.3. 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:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 IN [1, 2, 3]</td>
<td>true</td>
</tr>
<tr>
<td>2 IN [1, null, 3]</td>
<td>null</td>
</tr>
<tr>
<td>2 IN [1, 2, null]</td>
<td>true</td>
</tr>
<tr>
<td>2 IN [1]</td>
<td>false</td>
</tr>
<tr>
<td>2 IN []</td>
<td>false</td>
</tr>
<tr>
<td>null IN [1, 2, 3]</td>
<td>null</td>
</tr>
<tr>
<td>null IN [1, null, 3]</td>
<td>null</td>
</tr>
<tr>
<td>null IN []</td>
<td>false</td>
</tr>
</tbody>
</table>

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.

## 8.13.4. The [] operator and null

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1, 2, 3][null]</td>
<td>null</td>
</tr>
<tr>
<td>[1, 2, 3, 4][null..2]</td>
<td>null</td>
</tr>
<tr>
<td>[1, 2, 3][1..null]</td>
<td>null</td>
</tr>
<tr>
<td>{age: 25}[null]</td>
<td>null</td>
</tr>
</tbody>
</table>

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:

\[ a[\text{coalesce}(\text{lower}, 0) \ldots \text{coalesce}(\text{upper}, \text{size}(a))] \]

8.13.5. Expressions that return \textbf{null}

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

8.13.6. Using \textbf{IS NULL} and \textbf{IS NOT NULL}

Testing any value against \textbf{null}, either with the \(=\) operator or with the \(<\sim\) operator, always evaluates to \textbf{null}. Therefore, use the special equality operators \textbf{IS NULL} or \textbf{IS NOT NULL} instead (see \textit{Equality and comparison of values}).

[2] This is in accordance with the 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.
[3] The first week of any year is the week that contains the first Thursday of the year, and thus always contains January 4.
[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.
[5] Cypher does not support leap seconds; UTC-SLS (UTC with Smoothed Leap Seconds) is used to manage the difference in time between UTC and TAI (International Atomic Time).
[6] The expression \(\text{datetime().epochMillis}\) returns the equivalent value of the \(\text{timestamp()}\) function.
[7] For the nanosecond part of the epoch offset, the regular nanosecond component \(\text{instant.nanosecond}\) can be used.
Chapter 9. Clauses

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

- Administration clauses
- Importing data
- Listing functions and procedures
- Multiple graphs
- Projecting clauses
- Reading clauses
- Reading hints
- Reading sub-clauses
- Reading/Writing clauses
- Set operations
- Subquery clauses
- Transaction commands
- Writing clauses

Administration clauses

These comprise clauses used to manage databases, schema and security; further details can be found in Database management and Access control.

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE</td>
<td>Create, drop, start or stop a database.</td>
</tr>
<tr>
<td>DROP</td>
<td></td>
</tr>
<tr>
<td>START</td>
<td></td>
</tr>
<tr>
<td>STOP</td>
<td></td>
</tr>
<tr>
<td>DATABASE</td>
<td></td>
</tr>
<tr>
<td>INDEX</td>
<td>Create or drop an index on all nodes with a particular label and property.</td>
</tr>
<tr>
<td>CONSTRAINT</td>
<td>Create or drop a constraint pertaining to either a node label or relationship type, and a property.</td>
</tr>
</tbody>
</table>

Access control

Manage users, roles, and privileges for database, graph and sub-graph access control.

Importing data

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD CSV</td>
<td>Use when importing data from CSV files.</td>
</tr>
<tr>
<td>USING PERIODIC COMMIT</td>
<td>This query hint may be used to prevent an out-of-memory error from occurring when importing large amounts of data using LOAD CSV.</td>
</tr>
</tbody>
</table>
Listing functions and procedures

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHOW FUNCTIONS</td>
<td>List the available functions.</td>
</tr>
<tr>
<td>SHOW PROCEDURES</td>
<td>List the available procedures.</td>
</tr>
</tbody>
</table>

Multiple graphs

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USE</td>
<td>Determines which graph a query, or query part, is executed against.</td>
</tr>
</tbody>
</table>

Projecting clauses

These comprise clauses that define which expressions to return in the result set. The returned expressions may all be aliased using `AS`.

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RETURN ... [AS]</td>
<td>Defines what to include in the query result set.</td>
</tr>
<tr>
<td>WITH ... [AS]</td>
<td>Allows query parts to be chained together, piping the results from one to be used as starting points or criteria in the next.</td>
</tr>
<tr>
<td>UNWIND ... [AS]</td>
<td>Expands a list into a sequence of rows.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATCH</td>
<td>Specify the patterns to search for in the database.</td>
</tr>
<tr>
<td>OPTIONAL MATCH</td>
<td>Specify the patterns to search for in the database while using <code>nulls</code> for missing parts of the pattern.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Hint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USING INDEX</td>
<td>Index hints are used to specify which index, if any, the planner should use as a starting point.</td>
</tr>
<tr>
<td>Hint</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>USING INDEX SEEK</td>
<td>Index seek hint instructs the planner to use an index seek for this clause.</td>
</tr>
<tr>
<td>USING SCAN</td>
<td>Scan hints are used to force the planner to do a label scan (followed by a filtering operation) instead of using an index.</td>
</tr>
<tr>
<td>USING JOIN</td>
<td>Join hints are used to enforce a join operation at specified points.</td>
</tr>
</tbody>
</table>

**Reading sub-clauses**

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

<table>
<thead>
<tr>
<th>Sub-clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHERE</td>
<td>Adds constraints to the patterns in a MATCH or OPTIONAL MATCH clause or filters the results of a WITH clause.</td>
</tr>
<tr>
<td>WHERE EXISTS { ... }</td>
<td>An existential sub-query used to filter the results of a MATCH, OPTIONAL MATCH or WITH clause.</td>
</tr>
<tr>
<td>ORDER BY [ASC[ENDING]</td>
<td>DESC[ENDING]]</td>
</tr>
<tr>
<td>SKIP</td>
<td>Defines from which row to start including the rows in the output.</td>
</tr>
<tr>
<td>LIMIT</td>
<td>Constrains the number of rows in the output.</td>
</tr>
</tbody>
</table>

**Reading/Writing clauses**

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

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MERGE</td>
<td>Ensures that a pattern exists in the graph. Either the pattern already exists, or it needs to be created.</td>
</tr>
<tr>
<td>--- ON CREATE</td>
<td>Used in conjunction with MERGE, this write sub-clause specifies the actions to take if the pattern needs to be created.</td>
</tr>
<tr>
<td>--- ON MATCH</td>
<td>Used in conjunction with MERGE, this write sub-clause specifies the actions to take if the pattern already exists.</td>
</tr>
<tr>
<td>CALL ... [YIELD ... ]</td>
<td>Invokes a procedure deployed in the database and return any results.</td>
</tr>
</tbody>
</table>

**Set operations**
<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNION</td>
<td>Combines the result of multiple queries into a single result set. Duplicates are removed.</td>
</tr>
<tr>
<td>UNION ALL</td>
<td>Combines the result of multiple queries into a single result set. Duplicates are retained.</td>
</tr>
</tbody>
</table>

**Subquery clauses**

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL { ... }</td>
<td>Evaluates a subquery, typically used for post-union processing or aggregations.</td>
</tr>
<tr>
<td>CALL { ... } IN TRANSACTIONS</td>
<td>Evaluates a subquery in separate transactions. Typically used when modifying or importing large amounts of data.</td>
</tr>
</tbody>
</table>

**Transaction Commands**

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHOW TRANSACTIONS</td>
<td>List the available transactions.</td>
</tr>
<tr>
<td>TERMINATE TRANSACTIONS</td>
<td>Terminate transactions by their IDs.</td>
</tr>
</tbody>
</table>

**Writing clauses**

These comprise clauses that write the data to the database.

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE</td>
<td>Create nodes and relationships.</td>
</tr>
<tr>
<td>DELETE</td>
<td>Delete nodes, relationships or paths. Any node to be deleted must also have all associated relationships explicitly deleted.</td>
</tr>
<tr>
<td>DETACH DELETE</td>
<td>Delete a node or set of nodes. All associated relationships will automatically be deleted.</td>
</tr>
<tr>
<td>SET</td>
<td>Update labels on nodes and properties on nodes and relationships.</td>
</tr>
<tr>
<td>REMOVE</td>
<td>Remove properties and labels from nodes and relationships.</td>
</tr>
<tr>
<td>FOREACH</td>
<td>Update data within a list, whether components of a path, or the result of aggregation.</td>
</tr>
</tbody>
</table>
9.1. 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
  - Variable length relationships with multiple relationship types
  - 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
9.1.1. 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 paths. 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 for search performance, 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:

Graph

9.1.2. 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 88. Result
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 89. Result

<table>
<thead>
<tr>
<th>movie.title</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Wall Street&quot;</td>
</tr>
<tr>
<td>&quot;The American President&quot;</td>
</tr>
</tbody>
</table>

Rows: 2

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 90. Result

<table>
<thead>
<tr>
<th>movie.title</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Wall Street&quot;</td>
</tr>
</tbody>
</table>

Rows: 1
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 91. Result

<table>
<thead>
<tr>
<th>movie.title</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Wall Street&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

9.1.3. 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 92. Result

<table>
<thead>
<tr>
<th>movie.title</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Wall Street&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

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 93. Result
**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 94. Result**

<table>
<thead>
<tr>
<th>actor.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Michael Douglas&quot;</td>
</tr>
<tr>
<td>&quot;Martin Sheen&quot;</td>
</tr>
<tr>
<td>&quot;Charlie Sheen&quot;</td>
</tr>
</tbody>
</table>

Rows: 3

**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 95. Result**

<table>
<thead>
<tr>
<th>person.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Oliver Stone&quot;</td>
</tr>
<tr>
<td>&quot;Michael Douglas&quot;</td>
</tr>
<tr>
<td>&quot;Martin Sheen&quot;</td>
</tr>
<tr>
<td>&quot;Charlie Sheen&quot;</td>
</tr>
</tbody>
</table>

Rows: 4

**Match on relationship type and use a variable**

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

Query

```
MATCH (wallstreet {title: 'Wall Street'})<-[:ACTED_IN]-(actor)
RETURN r.role
```

Returns `ACTED_IN` roles for 'Wall Street'.

Table 96. Result

<table>
<thead>
<tr>
<th>r.role</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Gordon Gekko&quot;</td>
</tr>
<tr>
<td>&quot;Carl Fox&quot;</td>
</tr>
<tr>
<td>&quot;Bud Fox&quot;</td>
</tr>
</tbody>
</table>

Returns 3 rows.

9.1.4. Relationships in depth

Inside a single pattern, relationships will only be matched once. You can read more about this in Cypher path matching.

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 INCLUDING A SPACE]->(charlie)
```

Which leads to the following graph:

![Graph](image)

Query

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

Returns a relationship type with spaces in it.
Table 97. Result

<table>
<thead>
<tr>
<th>type(r)</th>
<th>&quot;TYPE INCLUDING A SPACE&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows:</td>
<td>1</td>
</tr>
</tbody>
</table>

Multiple relationships

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

Query

```sql
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 98. Result

<table>
<thead>
<tr>
<th>movie.title</th>
<th>director.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Wall Street&quot;</td>
<td>&quot;Oliver Stone&quot;</td>
</tr>
<tr>
<td>&quot;The American President&quot;</td>
<td>&quot;The American President&quot;</td>
</tr>
<tr>
<td>&quot;The American President&quot;</td>
<td>&quot;The American President&quot;</td>
</tr>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

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

```sql
MATCH (charlie {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 99. Result

<table>
<thead>
<tr>
<th>movie.title</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Wall Street&quot;</td>
</tr>
<tr>
<td>&quot;The American President&quot;</td>
</tr>
<tr>
<td>&quot;The American President&quot;</td>
</tr>
<tr>
<td>Rows: 3</td>
</tr>
</tbody>
</table>
Variable length relationships with multiple relationship types

Variable length relationships can be combined with multiple relationship types. In this case the *minHops..maxHops* applies to all relationship types as well as any combination of them.

Query

```
MATCH (charlie {name: 'Charlie Sheen'})-[[:ACTED_IN|DIRECTED]*2]-(person:Person)
RETURN person.name
```

Returns all people related to 'Charlie Sheen' by 2 hops with any combination of the relationship types ACTED_IN and DIRECTED.

Table 100. Result

<table>
<thead>
<tr>
<th>person.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Oliver Stone&quot;</td>
</tr>
<tr>
<td>&quot;Michael Douglas&quot;</td>
</tr>
<tr>
<td>&quot;Martin Sheen&quot;</td>
</tr>
</tbody>
</table>

Rows: 3

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 101. Result

<table>
<thead>
<tr>
<th>relationships(p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[:ACTED_IN[0]{role:&quot;Bud Fox&quot;},:ACTED_IN[2]{role:&quot;Gordon Gekko&quot;}]</td>
</tr>
<tr>
<td>[:ACTED_IN[0]{role:&quot;Bud Fox&quot;},:ACTED_IN[1]{role:&quot;Carl Fox&quot;}]</td>
</tr>
</tbody>
</table>

Rows: 2

Match with properties on a variable length path

A variable length relationship with properties defined on in it means that all relationships in the path must have the property set to the given value. In this query, there are two paths between 'Charlie Sheen' and his father 'Martin Sheen'. One of them includes a 'blocked' relationship and the other does not. In this case we first alter the original graph by using the following query to add BLOCKED and UNBLOCKED relationships:
This means that we are starting out with the following graph:

Table 102. Result

<table>
<thead>
<tr>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)-[X,7]-&gt;(7)&lt;-[X,8]-(1)</td>
</tr>
</tbody>
</table>

Rows: 1

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.

Table 103. Result

<table>
<thead>
<tr>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node[5]{title:&quot;Wall Street&quot;}</td>
</tr>
<tr>
<td>Node[3]{name:&quot;Oliver Stone&quot;}</td>
</tr>
<tr>
<td>Node[2]{name:&quot;Michael Douglas&quot;}</td>
</tr>
<tr>
<td>Node[1]{name:&quot;Martin Sheen&quot;}</td>
</tr>
<tr>
<td>Node[0]{name:&quot;Charlie Sheen&quot;}</td>
</tr>
</tbody>
</table>
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 104. Result

<table>
<thead>
<tr>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2)-[ACTED_IN,2]-&gt;(5)</td>
</tr>
<tr>
<td>(2)-[ACTED_IN,5]-&gt;(6)</td>
</tr>
</tbody>
</table>

Rows: 2

Matching on a bound relationship

When your pattern contains a bound relationship, and that relationship pattern does not 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 105. Result

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node[0]{name:&quot;Charlie Sheen&quot;}</td>
<td>Node[5]{title:&quot;Wall Street&quot;}</td>
</tr>
<tr>
<td>Node[5]{title:&quot;Wall Street&quot;}</td>
<td>Node[0]{name:&quot;Charlie Sheen&quot;}</td>
</tr>
</tbody>
</table>

Rows: 2

9.1.5. Shortest path

Single shortest path

Finding a single shortest path between two nodes is as easy as using the `shortestPath` function. It is done like this:
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 106. Result

<table>
<thead>
<tr>
<th>p</th>
<th>(1)-[ACTED_IN,1]-&gt;(5)&lt;-[DIRECTED,3]-(3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

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 do not consider the father/son relationship between the two.

Table 107. Result

<table>
<thead>
<tr>
<th>p</th>
<th>(0)-[ACTED_IN,0]-&gt;(5)&lt;-[ACTED_IN,1]-(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

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 108. Result

<table>
<thead>
<tr>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)-[ACTED_IN,1]-&gt;(5)&lt;-[ACTED_IN,2]-(2)</td>
</tr>
<tr>
<td>(1)-[ACTED_IN,4]-&gt;(6)&lt;-[ACTED_IN,5]-(2)</td>
</tr>
</tbody>
</table>

Rows: 2

9.1.6. Get node or relationship by id

Node by id

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

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

The corresponding node is returned.

Table 109. Result

<table>
<thead>
<tr>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node[0]{name:&quot;Charlie Sheen&quot;}</td>
</tr>
</tbody>
</table>

Rows: 1

Relationship by id

Search for relationships by id can be done with the `id()` function in a predicate. This is not the recommended practice. See Node by id for more information on the use of Neo4j ids.
The relationship with id 0 is returned.

<table>
<thead>
<tr>
<th>Table 110. Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
</tr>
<tr>
<td>‣ :ACTED_IN[0]{role:&quot;Bud Fox&quot;}</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

Multiple nodes by id

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

<table>
<thead>
<tr>
<th>Table 111. Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
</tr>
<tr>
<td>‣ Node[0]{name:&quot;Charlie Sheen&quot;}</td>
</tr>
<tr>
<td>‣ Node[3]{name:&quot;Oliver Stone&quot;}</td>
</tr>
<tr>
<td>‣ Node[5]{title:&quot;Wall Street&quot;}</td>
</tr>
<tr>
<td>Rows: 3</td>
</tr>
</tbody>
</table>

9.2. 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
9.2.1. 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:

![Graph](image)

9.2.2. 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 112. Result

<table>
<thead>
<tr>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

Rows: 1

9.2.3. Properties on optional elements

Returning a property from an optional element that is `null` will also return `null`.
9.2.4. 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)-->(x)
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>
<thead>
<tr>
<th>a.title</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Wall Street&quot;</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

Rows: 1

9.3. RETURN

The RETURN clause defines what to include in the query result set.
9.3.1. 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.

Graph

9.3.2. 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 115. Result

<table>
<thead>
<tr>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node[1]{name:&quot;B&quot;}</td>
</tr>
</tbody>
</table>

Rows: 1

9.3.3. 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 116. Result
9.3.4. 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 117. Result

<table>
<thead>
<tr>
<th>n.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;A&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

9.3.5. 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 118. Result

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>p</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node[0]{name:&quot;A&quot;, age:55, happy:&quot;Yes!&quot;}</td>
<td>Node[1]{name:&quot;B&quot;}</td>
<td>(0)-[BLOCKS,1]-&gt;(1)</td>
<td>:BLOCKS[1]{}</td>
</tr>
<tr>
<td>Node[0]{name:&quot;A&quot;, age:55, happy:&quot;Yes!&quot;}</td>
<td>Node[1]{name:&quot;B&quot;}</td>
<td>(0)-[KNOWS,0]-&gt;(1)</td>
<td>:KNOWS[0]{}</td>
</tr>
</tbody>
</table>

Rows: 2

9.3.6. 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:
The node with name "A" is returned.

Table 119. Result

<table>
<thead>
<tr>
<th>&quot;This isn't a common variable&quot;.happy</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Yes!&quot;</td>
</tr>
</tbody>
</table>

9.3.7. 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 120. Result

<table>
<thead>
<tr>
<th>SomethingTotallyDifferent</th>
</tr>
</thead>
<tbody>
<tr>
<td>55</td>
</tr>
</tbody>
</table>

9.3.8. 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 121. Result

<table>
<thead>
<tr>
<th>n.age</th>
</tr>
</thead>
<tbody>
<tr>
<td>55</td>
</tr>
<tr>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>
9.3.9. Other expressions

Any expression can be used as a return item — literals, predicates, properties, functions, and everything else.

Query

```cypher
MATCH (a {name: 'A'})
RETURN a.age > 30, "I'm a literal", [p=(a)-->(b) | p] AS `(a)-->()`
```

Returns a predicate, a literal and function call with a pattern expression parameter.

Table 122. Result

<table>
<thead>
<tr>
<th>a.age &gt; 30</th>
<th>&quot;I'm a literal&quot;</th>
<th>(a)--&gt;()</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>&quot;I'm a literal&quot;</td>
<td>[(0)-[BLOCKS,1]-&gt;(1),(0)-[KNOWS,0]-&gt;(1)]</td>
</tr>
</tbody>
</table>

Rows: 1

9.3.10. Unique results

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

Query

```cypher
MATCH (a {name: 'A'})-->(b)
RETURN DISTINCT b
```

The node named "B" is returned by the query, but only once.

Table 123. Result

<table>
<thead>
<tr>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node[1]{name:&quot;B&quot;}</td>
</tr>
</tbody>
</table>

Rows: 1

9.4. 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.
It is important to note that **WITH** affects variables in scope. Any variables not included in the **WITH** clause are not carried over to the rest of the query. The wildcard * can be used to include all variables that are currently in scope.

- **Introduction**
- **Introducing variables for expressions**
- **Using the wildcard to carry over variables**
- **Filter on aggregate function results**
- **Sort results before using collect on them**
- **Limit branching of a path search**

### 9.4.1. Introduction

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

One common usage of **WITH** is to limit the number of entries passed on to other **MATCH** clauses. By combining **ORDER BY** and **LIMIT**, it is possible to get the top X entries by some criteria and then bring in additional data from the graph.

**WITH** can also be used to introduce new variables containing the results of expressions for use in the following query parts (see **Introducing variables for expressions**). For convenience, the wildcard * expands to all variables that are currently in scope and carries them over to the next query part (see **Using the wildcard to carry over variables**).

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** 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.
9.4.2. Introducing variables for expressions

You can introduce new variables for the result of evaluating expressions.

Query

```
MATCH (george {name: 'George'})<-(otherPerson)
WITH otherPerson, toUpper(otherPerson.name) AS upperCaseName
WHERE upperCaseName STARTS WITH 'C'
RETURN otherPerson.name
```

This query returns the name of persons connected to 'George' whose name starts with a C, regardless of capitalization.

Table 124. Result

<table>
<thead>
<tr>
<th>otherPerson.name</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Caesar</em></td>
<td></td>
</tr>
</tbody>
</table>

Rows: 1

9.4.3. Using the wildcard to carry over variables

You can use the wildcard * to carry over all variables that are in scope, in addition to introducing new variables.

Query

```
MATCH (person)-[r]->(otherPerson)
WITH *, type(r) AS connectionType
RETURN person.name, otherPerson.name, connectionType
```

This query returns the names of all related persons and the type of relationship between them.

Table 125. Result

<table>
<thead>
<tr>
<th>person.name</th>
<th>otherPerson.name</th>
<th>connectionType</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;David&quot;</td>
<td>&quot;Anders&quot;</td>
<td>&quot;KNOWS&quot;</td>
</tr>
<tr>
<td>&quot;Anders&quot;</td>
<td>&quot;Bossman&quot;</td>
<td>&quot;KNOWS&quot;</td>
</tr>
<tr>
<td>&quot;Anders&quot;</td>
<td>&quot;Caesar&quot;</td>
<td>&quot;BLOCKS&quot;</td>
</tr>
<tr>
<td>&quot;Bossman&quot;</td>
<td>&quot;David&quot;</td>
<td>&quot;BLOCKS&quot;</td>
</tr>
<tr>
<td>&quot;Bossman&quot;</td>
<td>&quot;George&quot;</td>
<td>&quot;KNOWS&quot;</td>
</tr>
<tr>
<td>&quot;Caesar&quot;</td>
<td>&quot;George&quot;</td>
<td>&quot;KNOWS&quot;</td>
</tr>
</tbody>
</table>

Rows: 6

9.4.4. Filter on aggregate function results

Aggregated results have to pass through a WITH clause to be able to filter on.
The name of the person connected to 'David' with the at least more than one outgoing relationship will be returned by the query.

### Table 126. Result

<table>
<thead>
<tr>
<th>otherPerson.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Anders&quot;</td>
</tr>
</tbody>
</table>

**Rows:** 1

9.4.5. 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 127. Result

<table>
<thead>
<tr>
<th>collect(n.name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[&quot;George&quot;,&quot;David&quot;,&quot;Caesar&quot;]</td>
</tr>
</tbody>
</table>

**Rows:** 1

9.4.6. 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.
9.5. 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

9.5.1. 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.

9.5.2. 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 129. Result

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;val&quot;</td>
</tr>
</tbody>
</table>
9.5.3. Creating a distinct list

We want to transform a list of duplicates into a set using `DISTINCT`.

Query

```sql
WITH [1, 1, 2, 2] AS col1
UNWIND col1 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 130. Result

<table>
<thead>
<tr>
<th>setOfVals</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,2]</td>
</tr>
</tbody>
</table>

Rows: 1

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

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

Query

```sql
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 131. Result

<table>
<thead>
<tr>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

Rows: 4
9.5.5. Using UNWIND with a list of lists

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

Query

```sql
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 132. Result

<table>
<thead>
<tr>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

Rows: 5

9.5.6. 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

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

Table 133. Result

<table>
<thead>
<tr>
<th>(empty result)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 0</td>
</tr>
</tbody>
</table>

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

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

Using UNWIND on an expression that does not return a list, will return the same result as using UNWIND on a list that just contains that expression. As an example, UNWIND 5 is effectively equivalent to UNWIND[5]. 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 134. Result
(empty result)
Rows: 0

9.5.8. 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 135. Result

x
1
9.6. 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
  - Prefix string search using STARTS WITH
  - Suffix string search using ENDS WITH
  - Substring search using CONTAINS
  - 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
- Using existential subqueries in WHERE
  - Simple existential subquery
**Existential subquery with WHERE clause**

**Nesting existential subqueries**

- **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

- **Pattern element predicates**
  - Node pattern predicates

### 9.6.1. Introduction

WHERE is not a clause in its own right — rather, it’s part of MATCH, OPTIONAL MATCH and WITH.

In the case of WITH, 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.

<table>
<thead>
<tr>
<th>!</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>Indexes may be used to optimize queries using WHERE in a variety of cases.</td>
</tr>
</tbody>
</table>

The following graph is used for the examples below:
9.6.2. 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:Person)
WHERE n.name = 'Peter' XOR (n.age < 30 AND n.name = 'Timothy') OR NOT (n.name = 'Timothy' OR n.name = 'Peter')
RETURN n.name AS name, n.age AS age
ORDER BY name
```

Table 136. Result

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Andy&quot;</td>
<td>36</td>
</tr>
<tr>
<td>&quot;Peter&quot;</td>
<td>35</td>
</tr>
<tr>
<td>&quot;Timothy&quot;</td>
<td>25</td>
</tr>
</tbody>
</table>

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 'Andy' node will be returned.

Table 137. Result

<table>
<thead>
<tr>
<th>n.name</th>
<th>n.age</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Andy&quot;</td>
<td>36</td>
</tr>
</tbody>
</table>

Rows: 1

Filter on node property

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

Query

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

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

Table 138. Result

<table>
<thead>
<tr>
<th>n.name</th>
<th>n.age</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Timothy&quot;</td>
<td>25</td>
</tr>
</tbody>
</table>

Rows: 1

Filter on relationship property

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

Query

```
MATCH (n:Person)-[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 Andy has known him since before 2000.

Table 139. Result

<table>
<thead>
<tr>
<th>f.name</th>
<th>f.age</th>
<th>f.email</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Peter&quot;</td>
<td>35</td>
<td>&quot;<a href="mailto:peter_n@example.com">peter_n@example.com</a>&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

Filter on dynamically-computed node property

To filter on a property using a dynamically computed name, use square bracket syntax.
The name and age values for the 'Timothy' node are returned because he is less than 30 years of age.

Table 140. Result

<table>
<thead>
<tr>
<th>n.name</th>
<th>n.age</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Timothy&quot;</td>
<td>25</td>
</tr>
</tbody>
</table>

Rows: 1

Property existence checking

Use the IS NOT NULL predicate to only include nodes or relationships in which a property exists.

Query

```
MATCH (n:Person)
WHERE n.belt IS NOT NULL
RETURN n.name, n.belt
```

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

The `exists()` function has been deprecated for property existence checking and has been superseded by IS NOT NULL.

Table 141. Result

<table>
<thead>
<tr>
<th>n.name</th>
<th>n.belt</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Andy&quot;</td>
<td>&quot;white&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

9.6.3. String matching

The prefix and suffix of a string can be matched using STARTS WITH and ENDS WITH. To undertake a substring search - i.e. match regardless of location within a string - use CONTAINS. The matching is case-sensitive. Attempting to use these operators on values which are not strings will return null.

Prefix string search using STARTS WITH

The STARTS WITH operator is used to perform case-sensitive matching on the beginning of a string.

Query

```
MATCH (n:Person)
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 142. Result

<table>
<thead>
<tr>
<th>n.name</th>
<th>n.age</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Peter&quot;</td>
<td>35</td>
</tr>
</tbody>
</table>

Rows: 1

Suffix string search using **ENDS WITH**

The **ENDS WITH** operator is used to perform case-sensitive matching on the ending of a string.

Query

```
MATCH (n:Person)
WHERE n.name ENDsWith 'ter'
RETURN n.name, n.age
```

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

Table 143. Result

<table>
<thead>
<tr>
<th>n.name</th>
<th>n.age</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Peter&quot;</td>
<td>35</td>
</tr>
</tbody>
</table>

Rows: 1

Substring search using **CONTAINS**

The **CONTAINS** operator is used to perform case-sensitive matching regardless of location within a string.

Query

```
MATCH (n:Person)
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 144. Result

<table>
<thead>
<tr>
<th>n.name</th>
<th>n.age</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Peter&quot;</td>
<td>35</td>
</tr>
</tbody>
</table>

Rows: 1

String matching negation

Use the **NOT** keyword to exclude all matches on given string from your result:
The name and age for the 'Peter' node are returned because his name does not end with 'y'.

Table 145. Result

<table>
<thead>
<tr>
<th>n.name</th>
<th>n.age</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Peter&quot;</td>
<td>35</td>
</tr>
</tbody>
</table>

Rows: 1

9.6.4. Regular expressions

Cypher supports filtering using regular expressions. The regular expression syntax is inherited from the Java regular expressions. 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:Person)
WHERE n.name =~ 'Tim.*'
RETURN n.name, n.age

The name and age for the 'Timothy' node are returned because his name starts with 'Tim'.

Table 146. Result

<table>
<thead>
<tr>
<th>n.name</th>
<th>n.age</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Timothy&quot;</td>
<td>25</td>
</tr>
</tbody>
</table>

Rows: 1

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:Person)
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 147. Result

<table>
<thead>
<tr>
<th>n.name</th>
<th>n.age</th>
<th>n.email</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Peter&quot;</td>
<td>35</td>
<td>&quot;<a href="mailto:peter_n@example.com">peter_n@example.com</a>&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

Case-insensitive regular expressions

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

Query

```cypher
MATCH (n:Person)
WHERE n.name =~ '(?i)AND.*'
RETURN n.name, n.age
```

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

Table 148. Result

<table>
<thead>
<tr>
<th>n.name</th>
<th>n.age</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Andy&quot;</td>
<td>36</td>
</tr>
</tbody>
</table>

Rows: 1

9.6.5. 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 paths. MATCH (a)-[*]->(b) is very different from WHERE (a)-[*]->(b). The first will produce a path for every path it can find between a and b, whereas the latter will eliminate any matched paths where a and b do not have a directed relationship chain between them.

Query

```cypher
MATCH (timothy:Person {name: 'Timothy'}),
(other:Person)
WHERE other.name IN ['Andy', 'Peter'] AND (other)-->(timothy)
RETURN other.name, other.age
```
The name and age for nodes that have an outgoing relationship to the 'Timothy' node are returned.

Table 149. Result

<table>
<thead>
<tr>
<th>other.name</th>
<th>other.age</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Andy&quot;</td>
<td>36</td>
</tr>
</tbody>
</table>

Rows: 1

Filter on patterns using **NOT**

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

Query

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

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

Table 150. Result

<table>
<thead>
<tr>
<th>person.name</th>
<th>person.age</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Timothy&quot;</td>
<td>25</td>
</tr>
<tr>
<td>&quot;Peter&quot;</td>
<td>35</td>
</tr>
</tbody>
</table>

Rows: 2

Filter on patterns with properties

You can also add properties to your patterns:

Query

```
MATCH
  (n:Person)
WHERE (n)-[:KNOWS]-( (name: 'Timothy'))
RETURN n.name, n.age
```

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

Table 151. Result

<table>
<thead>
<tr>
<th>n.name</th>
<th>n.age</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Andy&quot;</td>
<td>36</td>
</tr>
</tbody>
</table>

Rows: 1

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:Person)-[r]->()
WHERE n.name='Andy' AND type(r) =~ 'K.*'
RETURN type(r), r.since
```

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

**Table 152. Result**

<table>
<thead>
<tr>
<th>type(r)</th>
<th>r.since</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;KNOWS&quot;</td>
<td>1999</td>
</tr>
<tr>
<td>&quot;KNOWS&quot;</td>
<td>2012</td>
</tr>
</tbody>
</table>

An existential subquery can be used to find out if a specified pattern exists at least once in the data. It can be used in the same way as a path pattern but it allows you to use MATCH and WHERE clauses internally. A subquery has a scope, as indicated by the opening and closing braces, { and }. Any variable that is defined in the outside scope can be referenced inside the subquery’s own scope. Variables introduced inside the subquery are not part of the outside scope and therefore can’t be accessed on the outside. If the subquery evaluates even once to anything that is not null, the whole expression will become true. This also means that the system only needs to calculate the first occurrence where the subquery evaluates to something that is not null and can skip the rest of the work.

**Syntax:**

```
EXISTS {
    MATCH [Pattern]
    WHERE [Expression]
}
```

It is worth noting that the MATCH keyword can be omitted in subqueries and that the WHERE clause is optional.

**9.6.6. Using existential subqueries in WHERE**

**Simple existential subquery**

Variables introduced by the outside scope can be used in the inner MATCH clause. The following example shows this:
Query

```cypher
MATCH (person:Person)
WHERE EXISTS (
  MATCH (person)-[:HAS_DOG]->(dog)
)
RETURN person.name AS name
```

Table 153. Result

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Andy&quot;</td>
</tr>
<tr>
<td>&quot;Peter&quot;</td>
</tr>
</tbody>
</table>

Rows: 2

Existential subquery with **WHERE** clause

A **WHERE** clause can be used in conjunction to the **MATCH**. Variables introduced by the **MATCH** clause and the outside scope can be used in this scope.

Query

```cypher
MATCH (person:Person)
WHERE EXISTS (  
  MATCH (person)-[:HAS_DOG]->(dog:Dog)  
  WHERE person.name = dog.name
)
RETURN person.name AS name
```

Table 154. Result

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Andy&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

Nesting existential subqueries

Existential subqueries can be nested like the following example shows. The nesting also affects the scopes. That means that it is possible to access all variables from inside the subquery which are either on the outside scope or defined in the very same subquery.

Query

```cypher
MATCH (person:Person)
WHERE EXISTS (  
  MATCH (person)-[:HAS_DOG]->(dog:Dog)  
  WHERE EXISTS (  
    MATCH (dog)-[:HAS_TOY]->(toy:Toy)  
    WHERE toy.name = 'Banana'
  )
)
RETURN person.name AS name
```

Table 155. Result

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

132
9.6.7. Lists

**IN operator**

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

**Query**

```sql
MATCH (a:Person)
WHERE a.name IN ['Peter', 'Timothy']
RETURN a.name, a.age
```

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

**Table 156. Result**

<table>
<thead>
<tr>
<th>a.name</th>
<th>a.age</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Timothy&quot;</td>
<td>25</td>
</tr>
<tr>
<td>&quot;Peter&quot;</td>
<td>35</td>
</tr>
</tbody>
</table>

Rows: 2

9.6.8. 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**

```sql
MATCH (n:Person)
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 157. Result**

<table>
<thead>
<tr>
<th>n.name</th>
<th>n.age</th>
<th>n.belt</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Andy&quot;</td>
<td>36</td>
<td>&quot;white&quot;</td>
</tr>
</tbody>
</table>

Rows: 1
Default to **true** if property is missing

If you want to compare a property on a node or relationship, but only if it exists, you can compare the property against both the value you are looking for and **null**, like:

**Query**

```sql
MATCH (n:Person)
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 158. Result**

<table>
<thead>
<tr>
<th>n.name</th>
<th>n.age</th>
<th>n.belt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andy</td>
<td>36</td>
<td>&quot;white&quot;</td>
</tr>
<tr>
<td>Peter</td>
<td>35</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>Timothy</td>
<td>25</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

Rows: 3

**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**

```sql
MATCH (person: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 159. Result**

<table>
<thead>
<tr>
<th>person.name</th>
<th>person.age</th>
<th>person.belt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peter</td>
<td>35</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

Rows: 1

**9.6.9. Using ranges**

**Simple range**

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

MATCH (a:Person)
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 160. Result

<table>
<thead>
<tr>
<th>a.name</th>
<th>a.age</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Timothy&quot;</td>
<td>25</td>
</tr>
<tr>
<td>&quot;Peter&quot;</td>
<td>35</td>
</tr>
</tbody>
</table>

Rows: 2

Composite range

Several inequalities can be used to construct a range.

Query

MATCH (a:Person)
WHERE a.name > 'Andy' AND a.name < 'Timothy'
RETURN a.name, a.age

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

Table 161. Result

<table>
<thead>
<tr>
<th>a.name</th>
<th>a.age</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Peter&quot;</td>
<td>35</td>
</tr>
</tbody>
</table>

Rows: 1

Pattern element predicates

Node pattern predicates

WHERE can appear inside a node pattern in a MATCH clause or a pattern comprehension:

Query

WITH 30 AS minAge
MATCH (a:Person WHERE a.name = 'Andy')-[[:KNOWS]->(b:Person WHERE b.age > minAge)
RETURN b.name

Table 162. Result
When used this way, predicates in \texttt{WHERE} can reference the node variable that the \texttt{WHERE} clause belongs to, but not other elements of the \texttt{MATCH} pattern.

The same rule applies to pattern comprehensions:

Query

\begin{verbatim}
MATCH (a:Person {name: 'Andy'})
RETURN ((a)-->(b WHERE b:Person) | b.name) AS friends
\end{verbatim}

Table 163. Result

<table>
<thead>
<tr>
<th>friends</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Peter&quot;,&quot;Timothy&quot;</td>
<td></td>
</tr>
</tbody>
</table>

\section*{9.7. ORDER BY}

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

- Introduction
- Order nodes by property
- Order nodes by multiple properties
- Order nodes by id
- Order nodes by expression
- Order nodes in descending order
- Ordering \texttt{null}
- Ordering in a \texttt{WITH} clause

\subsection*{9.7.1. Introduction}

\texttt{ORDER \ BY} relies on comparisons to sort the output, see \textit{Ordering and comparison of values}. You can sort on many different values, e.g. node/relationship properties, the node/relationship ids, or on most expressions. If you do not specify what to sort on, there is a risk that the results are arbitrarily sorted and therefore it is best practice to be specific when using \texttt{ORDER \ BY}.

In terms of scope of variables, \texttt{ORDER \ BY} follows special rules, depending on if the projecting \texttt{RETURN} or \texttt{WITH} clause is either aggregating or \texttt{DISTINCT}. If it is an aggregating or \texttt{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.

The performance of Cypher queries using ORDER BY on node properties can be influenced by the existence and use of an index for finding the nodes. If the index can provide the nodes in the order requested in the query, Cypher can avoid the use of an expensive Sort operation. Read more about this capability in The use of indexes.

The following graph is used for the examples below:

Strings that contain special characters can have inconsistent or non-deterministic ordering in Neo4j. For details, see Sorting of special characters.

9.7.2. 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 164. Result

<table>
<thead>
<tr>
<th>n.name</th>
<th>n.age</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;A&quot;</td>
<td>34</td>
</tr>
<tr>
<td>&quot;B&quot;</td>
<td>36</td>
</tr>
<tr>
<td>&quot;C&quot;</td>
<td>32</td>
</tr>
</tbody>
</table>

Rows: 3
9.7.3. 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 165. Result**

<table>
<thead>
<tr>
<th>n.name</th>
<th>n.age</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;C&quot;</td>
<td>32</td>
</tr>
<tr>
<td>&quot;A&quot;</td>
<td>34</td>
</tr>
<tr>
<td>&quot;B&quot;</td>
<td>36</td>
</tr>
</tbody>
</table>

Rows: 3

9.7.4. Order nodes by id

**ORDER BY** is used to sort the output.

**Query**

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

The nodes are returned, sorted by their internal id.

**Table 166. Result**

<table>
<thead>
<tr>
<th>n.name</th>
<th>n.age</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;A&quot;</td>
<td>34</td>
</tr>
<tr>
<td>&quot;B&quot;</td>
<td>36</td>
</tr>
<tr>
<td>&quot;C&quot;</td>
<td>32</td>
</tr>
</tbody>
</table>

Rows: 3

Keep in mind that 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 use application-generated ids instead.
9.7.5. Order nodes by expression

**ORDER BY** is used to sort the output.

Query

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

The nodes are returned, sorted by their properties.

**Table 167. Result**

<table>
<thead>
<tr>
<th>n.name</th>
<th>n.age</th>
<th>n.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;B&quot;</td>
<td>36</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;A&quot;</td>
<td>34</td>
<td>170</td>
</tr>
<tr>
<td>&quot;C&quot;</td>
<td>32</td>
<td>185</td>
</tr>
</tbody>
</table>

Rows: 3

9.7.6. 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 168. Result**

<table>
<thead>
<tr>
<th>n.name</th>
<th>n.age</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;C&quot;</td>
<td>32</td>
</tr>
<tr>
<td>&quot;B&quot;</td>
<td>36</td>
</tr>
<tr>
<td>&quot;A&quot;</td>
<td>34</td>
</tr>
</tbody>
</table>

Rows: 3

9.7.7. 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>
<thead>
<tr>
<th>n.length</th>
<th>n.name</th>
<th>n.age</th>
</tr>
</thead>
<tbody>
<tr>
<td>170</td>
<td>&quot;A&quot;</td>
<td>34</td>
</tr>
<tr>
<td>185</td>
<td>&quot;C&quot;</td>
<td>32</td>
</tr>
<tr>
<td>&lt;null&gt;</td>
<td>&quot;B&quot;</td>
<td>36</td>
</tr>
</tbody>
</table>

Table 169. Result
Rows: 3

9.7.8. Ordering in a WITH clause

When ORDER BY is present on a WITH clause, the immediately following clause will receive records in the specified order. The order is not guaranteed to be retained after the following clause, unless that also has an ORDER BY subclause. The ordering guarantee can be useful to exploit by operations which depend on the order in which they consume values. For example, this can be used to control the order of items in the list produced by the collect() aggregating function. The MERGE and SET clauses also have ordering dependencies which can be controlled this way.

Query

```
MATCH (n)
WITH n ORDER BY n.age
RETURN collect(n.name) AS names
```

The list of names built from the collect aggregating function contains the names in order of the age property.

Table 170. Result

<table>
<thead>
<tr>
<th>names</th>
</tr>
</thead>
<tbody>
<tr>
<td>[&quot;C&quot;,&quot;A&quot;,&quot;B&quot;]</td>
</tr>
</tbody>
</table>

Rows: 1

9.8. 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
9.8.1. 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.

![Graph](image)

9.8.2. Skip first three rows

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

**Query**

```cypher
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 171. Result**

<table>
<thead>
<tr>
<th>n.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;D&quot;</td>
</tr>
<tr>
<td>&quot;E&quot;</td>
</tr>
</tbody>
</table>

Rows: 2

9.8.3. Return middle two rows

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

**Query**

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

Two nodes from the middle are returned.

**Table 172. Result**
9.8.4. 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**

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

Skip the first row plus randomly 0, 1, or 2. So randomly skip 1, 2, or 3 rows.

**Table 173. Result**

<table>
<thead>
<tr>
<th>n.name</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;D&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;E&quot;</td>
<td></td>
</tr>
</tbody>
</table>

Rows: 2

---

9.9. **LIMIT**

**LIMIT** constrains the number of returned rows.

- **Introduction**
  - **Return a subset of the rows**
  - **Using an expression with **LIMIT** to return a subset of the rows**
  - **LIMIT** will not stop side effects

**9.9.1. Introduction**

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

9.9.2. Return a limited subset of the rows

To return a limited subset of the rows, use this syntax:

Query

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

Limit to 3 rows by the example query.

Table 174. Result

<table>
<thead>
<tr>
<th>n.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>“A”</td>
</tr>
<tr>
<td>“B”</td>
</tr>
<tr>
<td>“C”</td>
</tr>
</tbody>
</table>

Rows: 3

9.9.3. 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

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

Limit 1 row plus randomly 0, 1, or 2. So randomly limit to 1, 2, or 3 rows.

Table 175. Result

<table>
<thead>
<tr>
<th>n.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>“A”</td>
</tr>
</tbody>
</table>

Rows: 1
9.9.4. **LIMIT** will not stop side effects

The use of **LIMIT** in a query will not stop side effects, like **CREATE, DELETE** or **SET**, from happening if the limit is in the same query part as the side effect. This behaviour was undefined in versions before 4.3.

**Query**

```
CREATE (n)
RETURN n
LIMIT 0
```

This query returns nothing, but creates one node:

**Table 176. Result**

(Empty result)

Rows: 0
Nodes created: 1

**Query**

```
MATCH (n {name: 'A'})
SET n.age = 60
RETURN n
LIMIT 0
```

This query returns nothing, but writes one property:

**Table 177. Result**

(Empty result)

Rows: 0
Properties set: 1

If we want to limit the number of updates we can split the query using the **WITH** clause:

**Query**

```
MATCH (n)
WITH n LIMIT 1
SET n.locked = true
RETURN n
```

Writes **locked** property on one node and return that node:

**Table 178. Result**

```
<table>
<thead>
<tr>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node[0]{locked:true,name:&quot;A&quot;}</td>
</tr>
</tbody>
</table>
```

Rows: 1
Properties set: 1
9.10. CREATE

The `CREATE` clause is used to create 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.

9.10.1. Create nodes

Create single node

Creating a single node is done by issuing the following query:

**Query**

```
CREATE (n)
```

**Table 179. Result**

<table>
<thead>
<tr>
<th>(empty result)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 0</td>
</tr>
<tr>
<td>Nodes created: 1</td>
</tr>
</tbody>
</table>

Create multiple nodes

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

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

Table 180. Result

(\textit{empty result})

Rows: 0
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)
```

Table 181. Result

(\textit{empty result})

Rows: 0
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)
```

Table 182. Result

(\textit{empty result})

Rows: 0
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: 'Andy', title: 'Developer'})
```

Table 183. Result

(\textit{empty result})
Return created node

Creating a single node is done by issuing the following query:

Query

```sql
CREATE (a { name: 'Andy' })
RETURN a.name
```

The name of the newly-created node is returned.

Table 184. Result

<table>
<thead>
<tr>
<th>a.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Andy&quot;</td>
</tr>
</tbody>
</table>

Rows: 1
Nodes created: 1
Properties set: 1

9.10.2. 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

```sql
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 185. Result

<table>
<thead>
<tr>
<th>type(r)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;RELTYPE&quot;</td>
</tr>
</tbody>
</table>

Rows: 1
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 type and name of the newly-created relationship is returned by the example query.

Table 186. Result

<table>
<thead>
<tr>
<th>type(r)</th>
<th>r.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;RELTYPE&quot;</td>
<td>&quot;A&lt;-&gt;B&quot;</td>
</tr>
</tbody>
</table>

Rows: 1
Relationships created: 1
Properties set: 1

9.10.3. 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 = (andy {name: 'Andy'})-[::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 187. Result

<table>
<thead>
<tr>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2)-[::WORKS_AT,0]-&gt;(3)&lt;-[::WORKS_AT,1]-(4)</td>
</tr>
</tbody>
</table>

Rows: 1
Nodes created: 3
Relationships created: 2
Properties set: 2

9.10.4. 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

```json
{
  "props": {
    "name": "Andy",
    "position": "Developer"
  }
}
```

Query

```
CREATE (n:Person $props)
RETURN n
```

Table 188. Result

<table>
<thead>
<tr>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node[2]{name:&quot;Andy&quot;,position:&quot;Developer&quot;}</td>
</tr>
</tbody>
</table>

Rows: 1
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

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

Query

```
UNWIND $props AS map
CREATE (n)
SET n = map
```

Table 189. Result

<table>
<thead>
<tr>
<th>(empty result)</th>
</tr>
</thead>
</table>

Rows: 0
Nodes created: 2
Properties set: 4

9.11. DELETE

The DELETE clause is used to delete nodes, relationships or paths.
9.11.1. 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:

Graph

9.11.2. Delete single node

To delete a node, use the DELETE clause.

Query

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

Table 190. Result

(empty result)

Rows: 0
Nodes deleted: 1

9.11.3. Delete all nodes and relationships

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

Query

```cypher
MATCH (n)
DETACH DELETE n
```
9.11.4. 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: 'Andy'})
DETACH DELETE n
```

Table 192. Result

(empty result)
Rows: 0
Nodes deleted: 1
Relationships deleted: 2

For `DETACH DELETE` for users with restricted security privileges, see Operations Manual → Fine-grained access control.

9.11.5. Delete relationships only

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

Query

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

This deletes all outgoing `KNOWS` relationships from the node with the name 'Andy'.

Table 193. Result

(empty result)
Rows: 0
Relationships deleted: 2

9.12. SET

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

- Introduction
- Set a property
- Update a property
- Remove a property
- Copy properties between nodes and relationships
- Replace all properties using a map and =
- Remove all properties using an empty map and =
- Mutate specific properties using a map and +=
- Set multiple properties using one SET clause
- Set a property using a parameter
- Set all properties using a parameter
- Set a label on a node
- Set multiple labels on a node

9.12.1. Introduction

**SET** can be used with a map — provided as a literal, a parameter, or a node or relationship — to set properties.

Setting labels on a node is an idempotent operation — nothing will occur if an attempt is made to set a label on a node that already has that label. The query statistics will state whether any updates actually took place.

The examples use this graph as a starting point:

![Graph]

9.12.2. Set a property

Use **SET** to set a property on a node or relationship:
The newly-changed node is returned by the query.

Table 194. Result

<table>
<thead>
<tr>
<th>n.name</th>
<th>n.surname</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Andy&quot;</td>
<td>&quot;Taylor&quot;</td>
</tr>
</tbody>
</table>

Rows: 1
Properties set: 1

It is possible to set a property on a node or relationship using more complex expressions. For instance, in contrast to specifying the node directly, the following query shows how to set a property for a node selected by an expression:

Query

```
MATCH (n {name: 'Andy'})
SET (CASE WHEN n.age = 36 THEN n END).worksIn = 'Malmo'
RETURN n.name, n.worksIn
```

Table 195. Result

<table>
<thead>
<tr>
<th>n.name</th>
<th>n.worksIn</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Andy&quot;</td>
<td>&quot;Malmo&quot;</td>
</tr>
</tbody>
</table>

Rows: 1
Properties set: 1

No action will be taken if the node expression evaluates to null, as shown in this example:

Query

```
MATCH (n {name: 'Andy'})
SET (CASE WHEN n.age = 55 THEN n END).worksIn = 'Malmo'
RETURN n.name, n.worksIn
```

As no node matches the CASE expression, the expression returns a null. As a consequence, no updates occur, and therefore no worksIn property is set.

Table 196. Result

<table>
<thead>
<tr>
<th>n.name</th>
<th>n.worksIn</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Andy&quot;</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

Rows: 1
9.12.3. Update a property

`SET` can be used to update a property on a node or relationship. This query forces a change of type in the `age` property:

**Query**

```
MATCH (n {name: 'Andy'})
SET n.age = toString(n.age)
RETURN n.name, n.age
```

The `age` property has been converted to the string '36'.

**Table 197. Result**

<table>
<thead>
<tr>
<th>n.name</th>
<th>n.age</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Andy&quot;</td>
<td>&quot;36&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

Properties set: 1

9.12.4. Remove a property

Although `REMOVE` is normally used to remove a property, it's sometimes convenient to do it using the `SET` command. A case in point is if the property is provided by a parameter.

**Query**

```
MATCH (n {name: 'Andy'})
SET n.name = null
RETURN n.name, n.age
```

The `name` property is now missing.

**Table 198. Result**

<table>
<thead>
<tr>
<th>n.name</th>
<th>n.age</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;null&gt;</td>
<td>36</td>
</tr>
</tbody>
</table>

Rows: 1

Properties set: 1

9.12.5. Copy properties between nodes and relationships

`SET` can be used to copy all properties from one node or relationship to another. This will remove all other properties on the node or relationship being copied to.

**Query**

```
MATCH (at {name: 'Andy'}),(pn {name: 'Peter'})
SET at = pn
RETURN at.name, at.age, at.hungry, pn.name, pn.age
```

154
The 'Andy' node has had all its properties replaced by the properties of the 'Peter' node.

Table 199. Result

<table>
<thead>
<tr>
<th>at.name</th>
<th>at.age</th>
<th>at.hungry</th>
<th>pn.name</th>
<th>pn.age</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Peter&quot;</td>
<td>34</td>
<td>&lt;null&gt;</td>
<td>&quot;Peter&quot;</td>
<td>34</td>
</tr>
</tbody>
</table>

Rows: 1
Properties set: 3

9.12.6. Replace all properties using a map and =

The property replacement operator = can be used with SET to replace all existing properties on a node or relationship with those provided by a map:

Query

```
MATCH (p {name: 'Peter'})
SET p = {
  name: 'Peter Smith',
  position: 'Entrepreneur'
}
RETURN p.name, p.age, p.position
```

This query updated the name property from Peter to Peter Smith, deleted the age property, and added the position property to the 'Peter' node.

Table 200. Result

<table>
<thead>
<tr>
<th>p.name</th>
<th>p.age</th>
<th>p.position</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Peter Smith&quot;</td>
<td>&lt;null&gt;</td>
<td>&quot;Entrepreneur&quot;</td>
</tr>
</tbody>
</table>

Rows: 1
Properties set: 3

9.12.7. Remove all properties using an empty map and =

All existing properties can be removed from a node or relationship by using SET with = and an empty map as the right operand:

Query

```
MATCH (p {name: 'Peter'})
SET p = {}
RETURN p.name, p.age
```

This query removed all the existing properties — namely, name and age — from the 'Peter' node.

Table 201. Result

<table>
<thead>
<tr>
<th>p.name</th>
<th>p.age</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

Rows: 1
Properties set: 2
9.12.8. Mutate specific properties using a map and +=

The property mutation operator += can be used with SET to mutate properties from a map in a fine-grained fashion:

- Any properties in the map that are not on the node or relationship will be added.
- Any properties not in the map that are on the node or relationship will be left as is.
- Any properties that are in both the map and the node or relationship will be replaced in the node or relationship. However, if any property in the map is null, it will be removed from the node or relationship.

Query

```sql
MATCH (p {name: 'Peter'})
SET p += {age: 38, hungry: true, position: 'Entrepreneur'}
RETURN p.name, p.age, p.hungry, p.position
```

This query left the name property unchanged, updated the age property from 34 to 38, and added the hungry and position properties to the 'Peter' node.

Table 202. Result

<table>
<thead>
<tr>
<th>p.name</th>
<th>p.age</th>
<th>p.hungry</th>
<th>p.position</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Peter&quot;</td>
<td>38</td>
<td>true</td>
<td>&quot;Entrepreneur&quot;</td>
</tr>
</tbody>
</table>

Rows: 1
Properties set: 3

In contrast to the property replacement operator =, providing an empty map as the right operand to += will not remove any existing properties from a node or relationship. In line with the semantics detailed above, passing in an empty map with += will have no effect:

Query

```sql
MATCH (p {name: 'Peter'})
SET p += {}
RETURN p.name, p.age
```

Table 203. Result

<table>
<thead>
<tr>
<th>p.name</th>
<th>p.age</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Peter&quot;</td>
<td>34</td>
</tr>
</tbody>
</table>

Rows: 1

9.12.9. Set multiple properties using one SET clause

Set multiple properties at once by separating them with a comma:
9.12.10. Set a property using a parameter

Use a parameter to set the value of a property:

Parameters

```json
{
    "surname" : "Taylor"
}
```

Query

```cypher
MATCH (n { name: 'Andy'})
SET n.surname = $surname
RETURN n.name, n.surname
```

A `surname` property has been added to the 'Andy' node.

Table 205. Result

<table>
<thead>
<tr>
<th>n.name</th>
<th>n.surname</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Andy'</td>
<td>&quot;Taylor&quot;</td>
</tr>
</tbody>
</table>

Rows: 1
Properties set: 1

9.12.11. Set all properties using a parameter

This will replace all existing properties on the node with the new set provided by the parameter.

Parameters

```json
{
    "props" : {
        "name" : "Andy",
        "position" : "Developer"
    }
}
```

Query

```cypher
MATCH (n { name: 'Andy'})
SET n = $props
RETURN n.name, n.position, n.age, n.hungry
```
The 'Andy' node has had all its properties replaced by the properties in the `props` parameter.

Table 206. Result

<table>
<thead>
<tr>
<th>n.name</th>
<th>n.position</th>
<th>n.age</th>
<th>n.hungry</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Andy&quot;</td>
<td>&quot;Developer&quot;</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

Rows: 1
Properties set: 4

9.12.12. Set a label on a node

Use `SET` to set a label on a node:

**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 207. Result

<table>
<thead>
<tr>
<th>n.name</th>
<th>labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Stefan&quot;</td>
<td>[&quot;German&quot;]</td>
</tr>
</tbody>
</table>

Rows: 1
Labels added: 1

9.12.13. Set multiple labels on a node

Set multiple labels on a node with `SET` and use `:` to separate the different labels:

**Query**

```
MATCH (n {name: 'George'})
SET n:Swedish:Bossman
RETURN n.name, labels(n) AS labels
```

The newly-labeled node is returned by the query.

Table 208. Result

<table>
<thead>
<tr>
<th>n.name</th>
<th>labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;George&quot;</td>
<td>[&quot;Swedish&quot;,&quot;Bossman&quot;]</td>
</tr>
</tbody>
</table>

Rows: 1
Labels added: 2
The REMOVE clause is used to remove properties from nodes and relationships, and to remove labels from nodes.

- **Introduction**
- **Remove a property**
- **Remove all properties**
- **Remove a label from a node**
- **Remove multiple labels from a node**

### 9.13.1. 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:

<table>
<thead>
<tr>
<th>Swedish</th>
<th>age = 36</th>
<th>name = 'Andy'</th>
</tr>
</thead>
<tbody>
<tr>
<td>Swedish</td>
<td>KNOWS</td>
<td>KNOWS</td>
</tr>
<tr>
<td>German</td>
<td>age = 34</td>
<td>name = 'Peter'</td>
</tr>
<tr>
<td>Swedish</td>
<td>age = 25</td>
<td>name = 'Timothy'</td>
</tr>
</tbody>
</table>

### 9.13.2. 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: 'Andy'})
REMOVE a.age
RETURN a.name, a.age
```

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

**Table 209. Result**
9.13.3. Remove all properties

**REMOVE** cannot be used to remove all existing properties from a node or relationship. Instead, using **SET** with = and an empty map as the right operand will clear all properties from the node or relationship.

9.13.4. Remove a label from a node

To remove labels, you use **REMOVE**.

**Query**

```plaintext
MATCH (n {name: 'Peter'})
REMOVE n:German
RETURN n.name, labels(n)
```

**Table 210. Result**

<table>
<thead>
<tr>
<th>n.name</th>
<th>labels(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Peter&quot;</td>
<td>[&quot;Swedish&quot;]</td>
</tr>
</tbody>
</table>

Rows: 1
Labels removed: 1

9.13.5. Remove multiple labels from a node

To remove multiple labels, you use **REMOVE**.

**Query**

```plaintext
MATCH (n {name: 'Peter'})
REMOVE n:German:Swedish
RETURN n.name, labels(n)
```

**Table 211. Result**

<table>
<thead>
<tr>
<th>n.name</th>
<th>labels(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Peter&quot;</td>
<td>[]</td>
</tr>
</tbody>
</table>

Rows: 1
Labels removed: 2

9.14. FOREACH

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

Lists and paths are key concepts in Cypher. The `FOREACH` clause 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 — `SET`, `REMOVE`, `CREATE`, `MERGE`, `DELETE`, and `FOREACH`.

If you want to execute an additional `MATCH` for each element in a list then the `UNWIND` clause would be a more appropriate command.

![Graph](image)

9.14.2. Mark all nodes along a path

This query will set the property `marked` to `true` on all nodes along a path.

**Query**

```
MATCH p=(start)-[*]->(finish)
WHERE start.name = 'A' AND finish.name = 'D'
FOREACH (n IN nodes(p) | SET n.marked = true)
```

**Table 212. Result**

<table>
<thead>
<tr>
<th>(empty result)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 0</td>
</tr>
<tr>
<td>Properties set: 4</td>
</tr>
</tbody>
</table>
9.15. 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

9.15.1. 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.
For performance reasons, creating a schema index on the label or property is highly recommended when using `MERGE`. See `Indexes for search performance` for more information.

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 is all or nothing. If partial matches are needed, this can be accomplished by splitting a pattern up into multiple `MERGE` clauses.

Under concurrent updates, `MERGE` only guarantees existence of the `MERGE` pattern, but not uniqueness. To guarantee uniqueness of nodes with certain properties, a `unique constraint` should be used. See `Using unique constraints with MERGE` to see how `MERGE` can be used in combination with a unique constraint.

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

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

The following graph is used for the examples below:

```
Graph

9.15.2. 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.

<table>
<thead>
<tr>
<th>robert</th>
<th>labels(robert)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node[7]{}</td>
<td>[&quot;Critic&quot;]</td>
</tr>
</tbody>
</table>
Merge single node with properties

Merging a single node with properties where not all properties match any existing node.

Query

```mermaid
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.

Table 214. Result

<table>
<thead>
<tr>
<th>charlie</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node[7]{age:10,name:&quot;Charlie Sheen&quot;}</td>
</tr>
</tbody>
</table>

Rows: 1
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

```mermaid
MERGE (michael:Person {name: 'Michael Douglas'})
RETURN michael.name, michael.bornIn
```

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

Table 215. Result

<table>
<thead>
<tr>
<th>michael.name</th>
<th>michael.bornIn</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Michael Douglas&quot;</td>
<td>&quot;New Jersey&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

As mentioned previously, MERGE queries can greatly benefit from schema indexes. In this example, the following would significantly improve the performance of the MERGE clause:

```sql
CREATE INDEX PersonIndex FOR (n:Person) ON (n.name)
```
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

```plaintext
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.

Table 216. Result

<table>
<thead>
<tr>
<th>person.name</th>
<th>person.bornIn</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Charlie Sheen&quot;</td>
<td>&quot;New York&quot;</td>
<td>Node[7]{name:&quot;New York&quot;}</td>
</tr>
<tr>
<td>&quot;Martin Sheen&quot;</td>
<td>&quot;Ohio&quot;</td>
<td>Node[8]{name:&quot;Ohio&quot;}</td>
</tr>
<tr>
<td>&quot;Michael Douglas&quot;</td>
<td>&quot;New Jersey&quot;</td>
<td>Node[9]{name:&quot;New Jersey&quot;}</td>
</tr>
<tr>
<td>&quot;Oliver Stone&quot;</td>
<td>&quot;New York&quot;</td>
<td>Node[7]{name:&quot;New York&quot;}</td>
</tr>
<tr>
<td>&quot;Rob Reiner&quot;</td>
<td>&quot;New York&quot;</td>
<td>Node[7]{name:&quot;New York&quot;}</td>
</tr>
</tbody>
</table>

Rows: 5
Nodes created: 3
Properties set: 3
Labels added: 3

9.15.3. Use **ON CREATE** and **ON MATCH**

Merge with **ON CREATE**

Merge a node and set properties if the node needs to be created.

Query

```plaintext
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.

Table 217. Result

<table>
<thead>
<tr>
<th>keanu.name</th>
<th>keanu.created</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Keanu Reeves&quot;</td>
<td>1660050072989</td>
</tr>
</tbody>
</table>
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.

**Table 218. Result**

<table>
<thead>
<tr>
<th>person.name</th>
<th>person.found</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Charlie Sheen&quot;</td>
<td>true</td>
</tr>
<tr>
<td>&quot;Martin Sheen&quot;</td>
<td>true</td>
</tr>
<tr>
<td>&quot;Michael Douglas&quot;</td>
<td>true</td>
</tr>
<tr>
<td>&quot;Oliver Stone&quot;</td>
<td>true</td>
</tr>
<tr>
<td>&quot;Rob Reiner&quot;</td>
<td>true</td>
</tr>
</tbody>
</table>

Rows: 5
Properties set: 5

Merge with **ON CREATE** and **ON MATCH**

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.

**Table 219. Result**

<table>
<thead>
<tr>
<th>keanu.name</th>
<th>keanu.created</th>
<th>keanu.lastSeen</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Keanu Reeves&quot;</td>
<td>1660050076371</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>
Merge with ON MATCH setting multiple properties

If multiple properties should be set, simply separate them with commas.

Query

```mergl
MERGE (person:Person)
ON MATCH
SET
  person.found = true,
  person.lastAccessed = timestamp()
RETURN person.name, person.found, person.lastAccessed
```

Table 220. Result

<table>
<thead>
<tr>
<th>person.name</th>
<th>person.found</th>
<th>person.lastAccessed</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Charlie Sheen&quot;</td>
<td>true</td>
<td>1660050078169</td>
</tr>
<tr>
<td>&quot;Martin Sheen&quot;</td>
<td>true</td>
<td>1660050078169</td>
</tr>
<tr>
<td>&quot;Michael Douglas&quot;</td>
<td>true</td>
<td>1660050078169</td>
</tr>
<tr>
<td>&quot;Oliver Stone&quot;</td>
<td>true</td>
<td>1660050078169</td>
</tr>
<tr>
<td>&quot;Rob Reiner&quot;</td>
<td>true</td>
<td>1660050078169</td>
</tr>
</tbody>
</table>

Rows: 5
Properties set: 10

9.15.4. Merge relationships

Merge on a relationship

`MERGE` can be used to match or create a relationship.

Query

```mergl
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.

Table 221. Result
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.

**Table 222. Result**

<table>
<thead>
<tr>
<th>movie</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node[7]{ }</td>
</tr>
</tbody>
</table>

Rows: 1
Nodes created: 1
Relationships created: 2
Labels added: 1

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.

**Table 223. Result**

<table>
<thead>
<tr>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>:KNOWS[8]{}</td>
</tr>
</tbody>
</table>

Rows: 1
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').

**Table 224. Result**

<table>
<thead>
<tr>
<th>person.name</th>
<th>person.bornIn</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Charlie Sheen&quot;</td>
<td>&quot;New York&quot;</td>
<td>Node[7]{name: &quot;New York&quot;}</td>
</tr>
<tr>
<td>&quot;Martin Sheen&quot;</td>
<td>&quot;Ohio&quot;</td>
<td>Node[8]{name: &quot;Ohio&quot;}</td>
</tr>
<tr>
<td>&quot;Michael Douglas&quot;</td>
<td>&quot;New Jersey&quot;</td>
<td>Node[9]{name: &quot;New Jersey&quot;}</td>
</tr>
<tr>
<td>&quot;Oliver Stone&quot;</td>
<td>&quot;New York&quot;</td>
<td>Node[7]{name: &quot;New York&quot;}</td>
</tr>
<tr>
<td>&quot;Rob Reiner&quot;</td>
<td>&quot;New York&quot;</td>
<td>Node[7]{name: &quot;New York&quot;}</td>
</tr>
</tbody>
</table>

Rows: 5
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.

Table 225. Result

<table>
<thead>
<tr>
<th>person.name</th>
<th>person.chauffeurName</th>
<th>chauffeur</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Charlie Sheen&quot;</td>
<td>&quot;John Brown&quot;</td>
<td>Node[7]{name: &quot;John Brown&quot;}</td>
</tr>
<tr>
<td>&quot;Martin Sheen&quot;</td>
<td>&quot;Bob Brown&quot;</td>
<td>Node[8]{name: &quot;Bob Brown&quot;}</td>
</tr>
<tr>
<td>&quot;Michael Douglas&quot;</td>
<td>&quot;John Brown&quot;</td>
<td>Node[9]{name: &quot;John Brown&quot;}</td>
</tr>
<tr>
<td>&quot;Oliver Stone&quot;</td>
<td>&quot;Bill White&quot;</td>
<td>Node[10]{name: &quot;Bill White&quot;}</td>
</tr>
</tbody>
</table>

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

9.15.5. 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:

```cypher
CREATE CONSTRAINT FOR (n:Person) REQUIRE n.name IS UNIQUE;
CREATE CONSTRAINT FOR (n:Person) REQUIRE 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

```cypher
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.

Table 226. Result
Merge using unique constraints matches an existing node

Merge using unique constraints matches an existing node.

Query

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

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

<table>
<thead>
<tr>
<th>oliver.name</th>
<th>oliver.bornIn</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Oliver Stone&quot;</td>
<td>&quot;New York&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

Merge with unique constraints and partial matches

Merge using unique constraints fails when finding partial matches.

Query

```merlin
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

If we want to give Michael Douglas the role of Gordon Gekko, we can use the SET clause instead:

Query

```merlin
MERGE (michael:Person {name: 'Michael Douglas'})
SET michael.role = 'Gordon Gekko'
```
Merge with unique constraints and conflicting matches

Merge using unique constraints fails when finding conflicting matches.

Query

```mermaid
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
```

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

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

Table 228. Result

<table>
<thead>
<tr>
<th>person.name</th>
<th>person.role</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Keanu Reeves&quot;</td>
<td>&quot;Neo&quot;</td>
</tr>
</tbody>
</table>

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

9.16. CALL {} (subquery)

The `CALL {}` clause evaluates a subquery that returns some values.

- Introduction
9.16.1. Introduction

CALL allows to execute subqueries, i.e. queries inside of other queries. Subqueries allow you to compose queries, which is especially useful when working with `UNION` or aggregations.

Subqueries which end in a `RETURN` statement are called returning subqueries while subqueries without such a return statement are called unit subqueries.

A subquery is evaluated for each incoming input row. Every output row of a returning subquery is combined with the input row to build the result of the subquery. That means that a returning subquery will influence the number of rows. If the subquery does not return any rows, there will be no rows available after the subquery.

Unit subqueries on the other hand are called for their side-effects and not for their results and do therefore not influence the results of the enclosing query.

There are restrictions on how subqueries interact with the enclosing query:

- A subquery can only refer to variables from the enclosing query if they are explicitly imported.
- A subquery cannot return variables with the same names as variables in the enclosing query.
- All variables that are returned from a subquery are afterwards available in the enclosing query.

The following graph is used for the examples below:

```
Graph
```

9.16.2. Importing variables into subqueries

Variables are imported into a subquery using an importing `WITH` clause. As the subquery is evaluated for
each incoming input row, the imported variables get bound to the corresponding values from the input row in each evaluation.

Query

```cypher
UNWIND [0, 1, 2] AS x
CALL {
  WITH x
  RETURN x * 10 AS y
}
RETURN x, y
```

Table 229. Result

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
</tr>
</tbody>
</table>

Rows: 3

An importing WITH clause must:

- Consist only of simple references to outside variables - e.g. WITH x, y, z. Aliasing or expressions are not supported in importing WITH clauses - e.g. WITH a AS b or WITH a+1 AS b.
- Be the first clause of a subquery (or the second clause, if directly following a USE clause).

9.16.3. Post-union processing

Subqueries can be used to process the results of a UNION query further. This example query finds the youngest and the oldest person in the database and orders them by name.

Query

```cypher
CALL {
  MATCH (p:Person)
  RETURN p
  ORDER BY p.age ASC
  LIMIT 1
  UNION
  MATCH (p:Person)
  RETURN p
  ORDER BY p.age DESC
  LIMIT 1
}
RETURN p.name, p.age
ORDER BY p.name
```

Table 230. Result

<table>
<thead>
<tr>
<th>p.name</th>
<th>p.age</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Alice&quot;</td>
<td>20</td>
</tr>
<tr>
<td>&quot;Charlie&quot;</td>
<td>65</td>
</tr>
</tbody>
</table>

Rows: 2
If different parts of a result should be matched differently, with some aggregation over the whole results, subqueries need to be used. This example query finds friends and/or parents for each person. Subsequently the number of friends and parents are counted together.

**Query**

```
MATCH (p:Person)
CALL {
  WITH p
  OPTIONAL MATCH (p)-[:FRIEND_OF]->(other:Person)
  RETURN other
  UNION
  WITH p
  OPTIONAL MATCH (p)-[:CHILD_OF]->(other:Parent)
  RETURN other
}
RETURN DISTINCT p.name, count(other)
```

<table>
<thead>
<tr>
<th>p.name</th>
<th>count(other)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Alice&quot;</td>
<td>2</td>
</tr>
<tr>
<td>&quot;Bob&quot;</td>
<td>0</td>
</tr>
<tr>
<td>&quot;Charlie&quot;</td>
<td>0</td>
</tr>
<tr>
<td>&quot;Dora&quot;</td>
<td>0</td>
</tr>
</tbody>
</table>

Rows: 4

### 9.16.4. Aggregations

Returning subqueries change the number of results of the query: The result of the `CALL` clause is the combined result of evaluating the subquery for each input row.

The following example finds the name of each person and the names of their friends:

**Query**

```
MATCH (p:Person)
CALL {
  WITH p
  MATCH (p)-[:FRIEND_OF]->(c:Person)
  RETURN c.name AS friend
}
RETURN p.name, friend
```

<table>
<thead>
<tr>
<th>p.name</th>
<th>friend</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Alice&quot;</td>
<td>&quot;Bob&quot;</td>
</tr>
<tr>
<td>&quot;Bob&quot;</td>
<td>&quot;Alice&quot;</td>
</tr>
</tbody>
</table>

Rows: 2

The number of results of the subquery changed the number of results of the enclosing query: Instead of 4 rows, one for each node), there are now 2 rows which were found for Alice and Bob respectively. No rows
are returned for Charlie and Dora since they have no friends in our example graph.

We can also use subqueries to perform isolated aggregations. In this example we count the number of relationships each person has. As we get one row from each evaluation of the subquery, the number of rows is the same, before and after the CALL clause:

**Table 233. Result**

<table>
<thead>
<tr>
<th>p.name</th>
<th>numberOfConnections</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Alice&quot;</td>
<td>2</td>
</tr>
<tr>
<td>&quot;Bob&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;Charlie&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;Dora&quot;</td>
<td>0</td>
</tr>
</tbody>
</table>

Rows: 4

9.16.5. Unit subqueries and side-effects

Unit subqueries do not return any rows and are therefore used for their side effects.

This example query creates five clones of each existing person. As the subquery is a unit subquery, it does not change the number of rows of the enclosing query.

**Table 234. Result**

<table>
<thead>
<tr>
<th>count(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

Rows: 1
Nodes created: 20
Properties set: 20
Labels added: 20
9.16.6. Aggregation on imported variables

Aggregations in subqueries are scoped to the subquery evaluation, also for imported variables. The following example counts the number of younger persons for each person in the graph:

**Query**

```sql
MATCH (p:Person)
CALL {
    WITH p
    MATCH (other:Person)
    WHERE other.age < p.age
    RETURN count(other) AS youngerPersonsCount
}
RETURN p.name, youngerPersonsCount
```

<table>
<thead>
<tr>
<th>p.name</th>
<th>youngerPersonsCount</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Alice&quot;</td>
<td>0</td>
</tr>
<tr>
<td>&quot;Bob&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;Charlie&quot;</td>
<td>3</td>
</tr>
<tr>
<td>&quot;Dora&quot;</td>
<td>2</td>
</tr>
</tbody>
</table>

Rows: 4

9.16.7. Subqueries in transactions

Subqueries can be made to execute in separate, inner transactions, producing intermediate commits. This can come in handy when doing large write operations, like batch updates, imports, and deletes. To execute a subquery in separate transactions, you add the modifier `IN TRANSACTIONS` after the subquery.

The following example uses a CSV file and the `LOAD CSV` clause to import more data to the example graph. It creates nodes in separate transactions using `CALL {} IN TRANSACTIONS`:

**friends.csv**

```
1,Bill,26
2,Max,27
3,Anna,22
4,Gladys,29
5,Summer,24
```

**Query**

```sql
LOAD CSV FROM 'file:///friends.csv' AS line
CALL {
    WITH line
    CREATE (:PERSON {name: line[1], age: toInteger(line[2])})
} IN TRANSACTIONS
```

Table 236. Result

(Empty result)
As the size of the CSV file in this example is small, only a single separate transaction is started and committed.

**Batching**

The amount of work to do in each separate transaction can be specified in terms of how many input rows to process before committing the current transaction and starting a new one. The number of input rows is set with the modifier `OF n ROWS` (or `ROW`). If omitted, the default batch size is 1000 rows. The following is the same example but with one transaction every 2 input rows:

```
Query

LOAD CSV FROM 'file:///friends.csv' AS line
CALL {
  WITH line
  CREATE ( :Person { name: line[1], age: toInteger(line[2]) })
} IN TRANSACTIONS OF 2 ROWS
```

**Table 237. Result**

(Empty result)

```
Rows: 0
Nodes created: 5
Properties set: 10
Labels added: 5
Transactions committed: 3
```

The query now starts and commits three separate transactions:

1. The first two executions of the subquery (for the first two input rows from `LOAD CSV`) take place in the first transaction.
2. The first transaction is then committed before proceeding.
3. The next two executions of the subquery (for the next two input rows) take place in a second transaction.
4. The second transaction is committed.
5. The last execution of the subquery (for the last input row) takes place in a third transaction.
6. The third transaction is committed.

You can also use `CALL { ... } IN TRANSACTIONS OF n ROWS` to delete all your data in batches in order to avoid a huge garbage collection or an `OutOfMemory` exception. For example:
Up to a point, using a larger batch size will be more performant. The batch size of 2 ROWS is an example given the small data set used here. For larger data sets, you might want to use larger batch sizes, such as 10000 ROWS.

Errors

If an error occurs in CALL {} IN TRANSACTIONS the entire query fails and both the current inner transaction and the outer transaction are rolled back.

On error, any previously committed inner transactions remain committed, and are not rolled back.

In the following example, the last subquery execution in the second inner transaction fails due to division by zero.

Query

```cypher
UNWIND [4, 2, 1, 0] AS i
CALL {
  WITH i
  CREATE (:Example {num: 100/i})
} IN TRANSACTIONS OF 2 ROWS
RETURN i
```

/ by zero (Transactions committed: 1)

When the failure occurred, the first transaction had already been committed, so the database contains two example nodes

Query

```cypher
MATCH (e:Example)
RETURN e.num
```

Table 239. Result

Table 238. Result

(empty result)

Rows: 0
Nodes deleted: 9
Relationships deleted: 2
Transactions committed: 5
Restrictions

These are the restrictions on queries that use `CALL { ... } IN TRANSACTIONS`:

- A nested `CALL { ... } IN TRANSACTIONS` inside a `CALL { ... }` clause is not supported.
- A `CALL { ... } IN TRANSACTIONS` in a `UNION` is not supported.
- A `CALL { ... } IN TRANSACTIONS` after a write clause is not supported, unless that write clause is inside a `CALL { ... } IN TRANSACTIONS`.

9.17. CALL procedure

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

9.17.1. Introduction

Procedures are called using the `CALL` clause.

The `CALL` clause is also used to evaluate a subquery. For descriptions of the `CALL` clause in this context, refer to `CALL {} (subquery)`.

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` for explicit columns, 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). In a standalone procedure call, `YIELD *` can be used to select all columns. In this case, the name of the output parameters does not need to be known in advance.

For more information on how to determine the input parameters for the `CALL` procedure and the output parameters for the `YIELD` procedure, see View the signature for a procedure.

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

If the called procedure declares at least one result field, \texttt{YIELD} may generally not be omitted. However \texttt{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 \texttt{VOID} procedures. A \texttt{VOID} procedure is a procedure that does not declare any result fields and returns no result records and that has explicitly been declared as \texttt{VOID}. Calling a \texttt{VOID} procedure may only have a side effect and thus does neither allow nor require the use of \texttt{YIELD}. Calling a \texttt{VOID} procedure in the middle of a larger query will simply pass on each input record (i.e., it acts like \texttt{WITH \*} in terms of the record stream).

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Neo4j comes with a number of built-in procedures. For a list of these, see \textit{Operations Manual \to Procedures}.</td>
</tr>
<tr>
<td>Users can also develop custom procedures and deploy to the database. See \textit{Java Reference \to User-defined procedures} for details.</td>
</tr>
</tbody>
</table>

9.17.2. Call a procedure using \texttt{CALL}

This calls the built-in procedure \texttt{db.labels}, which lists all labels used in the database.

Query

\begin{lstlisting}[language=sq]
CALL db.labels()
\end{lstlisting}

Table 240. Result

<table>
<thead>
<tr>
<th>label</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>User</em></td>
</tr>
<tr>
<td><em>Administrator</em></td>
</tr>
</tbody>
</table>

Rows: 2

Cypher allows the omission of parentheses on procedures of arity-0 (no arguments).

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Best practice is to use parentheses for procedures.</td>
</tr>
</tbody>
</table>

Query

\begin{lstlisting}[language=sq]
CALL db.labels
\end{lstlisting}

Table 241. Result

<table>
<thead>
<tr>
<th>label</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>User</em></td>
</tr>
<tr>
<td><em>Administrator</em></td>
</tr>
</tbody>
</table>

Rows: 2
9.17.3. 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:

```
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`.

<table>
<thead>
<tr>
<th>Table 242. Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>signature</td>
</tr>
<tr>
<td>&quot;dbms.listConfig(searchString = :: STRING?) :: (name :: STRING?, description :: STRING?, value :: STRING?, dynamic :: BOOLEAN?)&quot;</td>
</tr>
</tbody>
</table>

| Rows: 1 |

9.17.4. 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.

```
CALL `db`.`labels()`
```

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

9.17.5. Call a procedure with literal arguments

This calls the example procedure `dbms.security.createUser` using literal arguments. The arguments are written out directly in the statement text.

```
CALL dbms.security.createUser('example_username', 'example_password', false)
```

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

9.17.6. Call a procedure with parameter arguments

This calls the example procedure `dbms.security.createUser` using parameters as arguments. Each procedure argument is taken to be the value of a corresponding statement parameter with the same name (or null if no such parameter has been given).
Examples that use parameter arguments shows the given parameters in JSON format; the exact manner in which they are to be submitted depends upon the driver being used. See Parameters, for more about querying with parameters.

Parameters

```
{
  "username" : "example_username",
  "password" : "example_password",
  "requirePasswordChange" : false
}
```

Query

```
CALL dbms.security.createUser($username, $password, $requirePasswordChange)
```

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

Cypher allows the omission of parentheses for procedures with arity-\(n\) (\(n\) arguments), Cypher implicitly passes the parameter arguments.

Best practice is to use parentheses for procedures. Omission of parantheses is available only in a so-called standalone procedure call, when the whole query consists of a single CALL clause.

Parameters

```
{
  "username" : "example_username",
  "password" : "example_password",
  "requirePasswordChange" : false
}
```

Query

```
CALL dbms.security.createUser
```

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

9.17.7. Call a procedure with mixed literal and parameter arguments

This calls the example procedure `dbms.security.createUser` using both literal and parameter arguments.

Parameters

```
{
  "password" : "example_password"
}
```

Query

```
CALL dbms.security.createUser('example_username', $password, false)
```
Since our example procedure does not return any result, the result is empty.

9.17.8. Call a procedure with literal and default arguments

This calls the example procedure `dbms.security.createUser` using literal arguments. That is, arguments that are written out directly in the statement text, and a trailing default argument that is provided by the procedure itself.

Query

```sql
CALL dbms.security.createUser('example_username', 'example_password')
```

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

9.17.9. Call a procedure using `CALL YIELD *`

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

Query

```sql
CALL db.labels() YIELD *
```

If the procedure has deprecated return columns, those columns are also returned.

9.17.10. 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

```sql
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.

9.17.11. 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

```sql
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.
9.17.12. 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

```sql
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.

9.18. 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

9.18.1. 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.

Graph
9.18.2. Combine two queries and retain duplicates

Combining the results from two queries is done using `UNION ALL`.

**Query**

```plaintext
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 243. Result**

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Anthony Hopkins&quot;</td>
</tr>
<tr>
<td>&quot;Helen Mirren&quot;</td>
</tr>
<tr>
<td>&quot;Hitchcock&quot;</td>
</tr>
<tr>
<td>&quot;Hitchcock&quot;</td>
</tr>
</tbody>
</table>

Rows: 4

9.18.3. Combine two queries and remove duplicates

By not including `ALL` in the `UNION`, duplicates are removed from the combined result set.

**Query**

```plaintext
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 244. Result**

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Anthony Hopkins&quot;</td>
</tr>
<tr>
<td>&quot;Helen Mirren&quot;</td>
</tr>
<tr>
<td>&quot;Hitchcock&quot;</td>
</tr>
</tbody>
</table>

Rows: 3

9.19. USE

The `USE` clause determines which graph a query, or query part, is executed against.
9.19.1. Introduction

The **USE** clause determines which graph a query, or query part, is executed against. It is supported for queries and schema commands.

The **USE** clause can not be used together with the **PERIODIC COMMIT** clause.

9.19.2. Syntax

The **USE** clause can only appear as the prefix of schema commands, or as the first clause of queries:

```
USE <graph> <other clauses>
```

Where `<graph>` refers to the name or alias of a database in the DBMS.

**Fabric syntax**

When running queries against a Fabric database, the **USE** clause can also appear as the first clause of:

- Union parts:
  ```
  USE <graph> <other clauses>
  UNION
  USE <graph> <other clauses>
  ```

- Subqueries:
  ```
  CALL {
  USE <graph> <other clauses>
  }
  ```

In subqueries, a **USE** clause may appear as the second clause, if directly following an importing **WITH**

When executing queries against a Fabric database, in addition to referring to databases in the DBMS, the `<graph>` may also refer to a graph mounted through the Fabric configuration. For more information, see Operations Manual → Fabric.
9.19.3. Examples

Query a graph by name

In this example we assume that your DBMS contains a database named `myDatabase`:

```
USE myDatabase
MATCH (n) RETURN n
```

Query a Fabric graph by name

In this example we assume that we have configured a Fabric database called `exampleFabricSetup`. The graph that we wish to query is named `exampleDatabaseName`:

```
USE exampleFabricSetup.exampleDatabaseName
MATCH (n) RETURN n
```

Query a Fabric graph by graph ID

This examples continues with a Fabric database called `exampleFabricSetup`.

The graph we wish to query is configured with the graph id `0`, which is why we can refer to it using the built-in function `graph()` with the argument `0`:

```
USE exampleFabricSetup.graph(0)
MATCH (n) RETURN n
```

9.20. 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 remote 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
• Using linenumber() with LOAD CSV
• Using file() with LOAD CSV

9.20.1. 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.
• 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 supports resources compressed with gzip and Deflate. Additionally LOAD CSV supports locally stored CSV files compressed with ZIP.
• 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 [deprecated]#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 should be set to a single directory relative to the Neo4j installation path on the database server. All requests to load from file:/// URLs will then be relative to the specified directory. The default value set in the config settings is import. This is a security measure which prevents the database from accessing files outside the standard import directory, similar to how a Unix chroot operates. Setting this to an empty field will allow access to all files within the Neo4j installation folder. Commenting out this setting will disable the security feature, allowing all files in the local system to be imported. This is definitely 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.

• When using file:/// URLs, spaces and other non-alphanumeric characters need to be URL encoded.

• 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 <NEO4J_HOME>/data/csv/myfile.csv and <NEO4J_HOME>/data/csv/myproject/myfile.csv respectively.
The file location is relative to the import. The config setting `dbms.directories.import` only applies to local disc and not to remote URLs.

See the examples below for further details.

9.20.2. 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 ";
- if `dbms.import.csv.legacy_quote_escaping` is set to the default value of true, \ is used as an escape character;
- a double quote must be in a quoted string and escaped, either with the escape character or a second double quote.

9.20.3. 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 'file:///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
```
9.20.4. Import data from a remote CSV file

Accordingly, you can import data from a CSV file in a remote location into Neo4j. Note that this applies to all variations of CSV files (see examples below for other variations).

data.neo4j.com/bands/artists.csv

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ABBA, 1992</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Roxette, 1986</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Europe, 1979</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>The Cardigans, 1992</td>
<td></td>
</tr>
</tbody>
</table>

Query

```
LOAD CSV FROM 'https://data.neo4j.com/bands/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
```

9.20.5. 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

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ABBA, 1992</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Roxette, 1986</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Europe, 1979</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>The Cardigans, 1992</td>
<td></td>
</tr>
</tbody>
</table>

Query

```
LOAD CSV WITH HEADERS FROM 'file:///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
```
9.20.6. 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, \u003B is equivalent to ; (SEMICOLON).

artists-fieldterminator.csv

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

Query

```
LOAD CSV FROM 'file:///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.

9.20.7. 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 happens every 1000 rows. Note that PERIODIC COMMIT is only allowed in implicit (auto-commit or :auto) transactions. For more information, see [deprecated PERIODIC COMMIT query hint](#).

Note: The USE clause can not be used together with the PERIODIC COMMIT query hint.

Note: Queries with the PERIODIC COMMIT query hint can not be routed by Server-side routing.

Query

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

9.20.9. 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 'file:///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!
9.20.10. Using linenumber() with LOAD CSV

For certain scenarios, like debugging a problem with a csv file, it may be useful to get the current line number that LOAD CSV is operating on. The linenumber() function provides exactly that or null if called without a LOAD CSV context.

```
RESULT
<table>
<thead>
<tr>
<th>name</th>
<th>year</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;The &quot;Symbol&quot;&quot;</td>
<td>1992</td>
<td>12</td>
</tr>
</tbody>
</table>
1 row
Nodes created: 1
Properties set: 2
Labels added: 1
```

9.20.11. Using file() with LOAD CSV

For certain scenarios, like debugging a problem with a csv file, it may be useful to get the absolute path of the file that LOAD CSV is operating on. The file() function provides exactly that or null if called without a LOAD CSV context.

```
artists.csv
1,ABBA,1992  
2,Roxette,1986 
3,Europe,1979 
4,The Cardigans,1992 
```

Query

```
LOAD CSV FROM 'file:///artists.csv' AS line
RETURN linenumber() AS number, line
```

```
RESULT
<table>
<thead>
<tr>
<th>number</th>
<th>line</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[&quot;1&quot;,&quot;ABBA&quot;,&quot;1992&quot;]</td>
</tr>
<tr>
<td>2</td>
<td>[&quot;2&quot;,&quot;Roxette&quot;,&quot;1986&quot;]</td>
</tr>
<tr>
<td>3</td>
<td>[&quot;3&quot;,&quot;Europe&quot;,&quot;1979&quot;]</td>
</tr>
<tr>
<td>4</td>
<td>[&quot;4&quot;,&quot;The Cardigans&quot;,&quot;1992&quot;]</td>
</tr>
</tbody>
</table>
4 rows
```

```
artists.csv
1,ABBA,1992  
2,Roxette,1986 
3,Europe,1979 
4,The Cardigans,1992 
```

194
Since `LOAD CSV` can temporarily download a file to process it, it is important to note that `file()` will always return the path on disk. If `LOAD CSV` is invoked with a `file:///` URL that points to your disk `file()` will return that same path.

### Result

<table>
<thead>
<tr>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;/home/example/neo4j/import/artists.csv&quot;</td>
</tr>
</tbody>
</table>

1 row

### 9.21. SHOW FUNCTIONS

This section explains the `SHOW FUNCTIONS` command.

Listing the available functions can be done with `SHOW FUNCTIONS`.

The command `SHOW FUNCTIONS` returns only the default output. For a full output use the optional `YIELD` command. Full output: `SHOW FUNCTIONS YIELD *`.

This command will produce a table with the following columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name of the function.</td>
</tr>
<tr>
<td>category</td>
<td>The function category, for example <code>scalar</code> or <code>string</code>.</td>
</tr>
<tr>
<td>description</td>
<td>The function description.</td>
</tr>
<tr>
<td>signature</td>
<td>The signature of the function.</td>
</tr>
<tr>
<td>isBuiltIn</td>
<td>Whether the function is built-in or user-defined.</td>
</tr>
<tr>
<td>argumentDescription</td>
<td>List of the arguments for the function, as map of strings with name, type, default, and description.</td>
</tr>
</tbody>
</table>
9.21.1. Syntax

List functions, either all or only built-in or user-defined

```sql
SHOW [ALL|BUILT IN|USER DEFINED] FUNCTION[S]
[YIELD ( * | field[, ...] ) [ORDER BY field[, ...]] [SKIP n] [LIMIT n]]
[WHERE expression]
[RETURN field[, ...] [ORDER BY field[, ...]] [SKIP n] [LIMIT n]]
```

When using the RETURN clause, the YIELD clause is mandatory and must not be omitted.

List functions that the current user can execute

```sql
SHOW [ALL|BUILT IN|USER DEFINED] FUNCTION[S] EXECUTABLE [BY CURRENT USER]
[YIELD ( * | field[, ...] ) [ORDER BY field[, ...]] [SKIP n] [LIMIT n]]
[WHERE expression]
[RETURN field[, ...] [ORDER BY field[, ...]] [SKIP n] [LIMIT n]]
```

When using the RETURN clause, the YIELD clause is mandatory and must not be omitted.

List functions that the specified user can execute

```sql
SHOW [ALL|BUILT IN|USER DEFINED] FUNCTION[S] EXECUTABLE BY username
[YIELD ( * | field[, ...] ) [ORDER BY field[, ...]] [SKIP n] [LIMIT n]]
[WHERE expression]
[RETURN field[, ...] [ORDER BY field[, ...]] [SKIP n] [LIMIT n]]
```

Required privilege `SHOW USER`. This command cannot be used for LDAP users.

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>returnDescription</td>
<td>The return value type.</td>
</tr>
<tr>
<td>aggregating</td>
<td>Whether the function is aggregating or not.</td>
</tr>
<tr>
<td>rolesExecution</td>
<td>List of roles permitted to execute this function. Is null without the SHOW ROLE privilege.</td>
</tr>
<tr>
<td>rolesBoostedExecution</td>
<td>List of roles permitted to use boosted mode when executing this function. Is null without the SHOW ROLE privilege.</td>
</tr>
</tbody>
</table>

9.21.2. Listing all functions

To list all available functions with the default output columns, the `SHOW FUNCTIONS` command can be used. If all columns are required, use `SHOW FUNCTIONS YIELD *`. 
### Table 246. Result

<table>
<thead>
<tr>
<th>name</th>
<th>category</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;abs&quot;</td>
<td>&quot;Numeric&quot;</td>
<td>&quot;Returns the absolute value of an integer.&quot;</td>
</tr>
<tr>
<td>&quot;abs&quot;</td>
<td>&quot;Numeric&quot;</td>
<td>&quot;Returns the absolute value of a floating point number.&quot;</td>
</tr>
<tr>
<td>&quot;acos&quot;</td>
<td>&quot;Trigonometric&quot;</td>
<td>&quot;Returns the arccosine of a number in radians.&quot;</td>
</tr>
<tr>
<td>&quot;all&quot;</td>
<td>&quot;Predicate&quot;</td>
<td>&quot;Returns true if the predicate holds for all elements in the given list.&quot;</td>
</tr>
<tr>
<td>&quot;any&quot;</td>
<td>&quot;Predicate&quot;</td>
<td>&quot;Returns true if the predicate holds for at least one element in the given list.&quot;</td>
</tr>
<tr>
<td>&quot;asin&quot;</td>
<td>&quot;Trigonometric&quot;</td>
<td>&quot;Returns the arcsine of a number in radians.&quot;</td>
</tr>
<tr>
<td>&quot;atan&quot;</td>
<td>&quot;Trigonometric&quot;</td>
<td>&quot;Returns the arctangent of a number in radians.&quot;</td>
</tr>
<tr>
<td>&quot;atan2&quot;</td>
<td>&quot;Trigonometric&quot;</td>
<td>&quot;Returns the arctangent2 of a set of coordinates in radians.&quot;</td>
</tr>
<tr>
<td>&quot;avg&quot;</td>
<td>&quot;Aggregating&quot;</td>
<td>&quot;Returns the average of a set of integer values.&quot;</td>
</tr>
<tr>
<td>&quot;avg&quot;</td>
<td>&quot;Aggregating&quot;</td>
<td>&quot;Returns the average of a set of floating point values.&quot;</td>
</tr>
<tr>
<td>&quot;avg&quot;</td>
<td>&quot;Aggregating&quot;</td>
<td>&quot;Returns the average of a set of duration values.&quot;</td>
</tr>
<tr>
<td>&quot;ceil&quot;</td>
<td>&quot;Numeric&quot;</td>
<td>&quot;Returns the smallest floating point number that is greater than or equal to a number and equal to a mathematical integer.&quot;</td>
</tr>
<tr>
<td>&quot;coalesce&quot;</td>
<td>&quot;Scalar&quot;</td>
<td>&quot;Returns the first non-null value in a list of expressions.&quot;</td>
</tr>
<tr>
<td>&quot;collect&quot;</td>
<td>&quot;Aggregating&quot;</td>
<td>&quot;Returns a list containing the values returned by an expression.&quot;</td>
</tr>
<tr>
<td>&quot;cos&quot;</td>
<td>&quot;Trigonometric&quot;</td>
<td>&quot;Returns the cosine of a number.&quot;</td>
</tr>
<tr>
<td>&quot;cot&quot;</td>
<td>&quot;Trigonometric&quot;</td>
<td>&quot;Returns the cotangent of a number.&quot;</td>
</tr>
<tr>
<td>&quot;count&quot;</td>
<td>&quot;Aggregating&quot;</td>
<td>&quot;Returns the number of values or rows.&quot;</td>
</tr>
<tr>
<td>&quot;date&quot;</td>
<td>&quot;Temporal&quot;</td>
<td>&quot;Create a Date instant.&quot;</td>
</tr>
<tr>
<td>&quot;date.realtime&quot;</td>
<td>&quot;Temporal&quot;</td>
<td>&quot;Get the current Date instant using the realtime clock.&quot;</td>
</tr>
<tr>
<td>&quot;date.statement&quot;</td>
<td>&quot;Temporal&quot;</td>
<td>&quot;Get the current Date instant using the statement clock.&quot;</td>
</tr>
</tbody>
</table>
9.21.3. Listing functions with filtering on output columns

The listed functions can be filtered in multiple ways. One way is through the type keywords, **BUILT IN** and **USER DEFINED**. A more flexible way is to use the **WHERE** clause. For example, getting the name of all built-in functions starting with the letter 'a':

**Query**

```
SHOW BUILT IN FUNCTIONS YIELD name, isBuiltIn
WHERE name STARTS WITH 'a'
```

<table>
<thead>
<tr>
<th>name</th>
<th>isBuiltIn</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>true</td>
</tr>
<tr>
<td>abs</td>
<td>true</td>
</tr>
<tr>
<td>acos</td>
<td>true</td>
</tr>
<tr>
<td>all</td>
<td>true</td>
</tr>
<tr>
<td>any</td>
<td>true</td>
</tr>
<tr>
<td>asin</td>
<td>true</td>
</tr>
<tr>
<td>atan</td>
<td>true</td>
</tr>
<tr>
<td>atan2</td>
<td>true</td>
</tr>
<tr>
<td>avg</td>
<td>true</td>
</tr>
<tr>
<td>avg</td>
<td>true</td>
</tr>
<tr>
<td>avg</td>
<td>true</td>
</tr>
</tbody>
</table>

Table 247. Result

9.21.4. Listing functions with other filtering

The listed functions can also be filtered on whether a user can execute them. This filtering is only available through the **EXECUTABLE** clause and not through the **WHERE** clause. This is due to using the user’s privileges instead of filtering on the available output columns.

There are two options, how to use the **EXECUTABLE** clause. The first option, is to filter for the current user:

**Query**

```
SHOW FUNCTIONS EXECUTABLE BY CURRENT USER YIELD *
```

<table>
<thead>
<tr>
<th>name</th>
<th>category</th>
<th>description</th>
<th>rolesExecution</th>
<th>rolesBoostedExecution</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>Numeric</td>
<td>&quot;Returns the absolute value of an integer.&quot;</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
<td></td>
</tr>
</tbody>
</table>

Table 248. Result
<table>
<thead>
<tr>
<th>name</th>
<th>category</th>
<th>description</th>
<th>rolesExecution</th>
<th>rolesBoostedExecution</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;abs&quot;</td>
<td>&quot;Numeric&quot;</td>
<td>&quot;Returns the absolute value of a floating point number.&quot;</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;acos&quot;</td>
<td>&quot;Trigonometric&quot;</td>
<td>&quot;Returns the arccosine of a number in radians.&quot;</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;all&quot;</td>
<td>&quot;Predicate&quot;</td>
<td>&quot;Returns true if the predicate holds for all elements in the given list.&quot;</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;any&quot;</td>
<td>&quot;Predicate&quot;</td>
<td>&quot;Returns true if the predicate holds for at least one element in the given list.&quot;</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;asin&quot;</td>
<td>&quot;Trigonometric&quot;</td>
<td>&quot;Returns the arcsine of a number in radians.&quot;</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;atan&quot;</td>
<td>&quot;Trigonometric&quot;</td>
<td>&quot;Returns the arctangent of a number in radians.&quot;</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;atan2&quot;</td>
<td>&quot;Trigonometric&quot;</td>
<td>&quot;Returns the arctangent2 of a set of coordinates in radians.&quot;</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;avg&quot;</td>
<td>&quot;Aggregating&quot;</td>
<td>&quot;Returns the average of a set of integer values.&quot;</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;avg&quot;</td>
<td>&quot;Aggregating&quot;</td>
<td>&quot;Returns the average of a set of floating point values.&quot;</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

Rows: 10

Notice that the two roles columns are empty due to missing the SHOW ROLE privilege.

The second option, is to filter for a specific user:

Query

```
SHOW FUNCTIONS EXECUTABLE BY jake
```

Table 249. Result
<table>
<thead>
<tr>
<th>name</th>
<th>category</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;abs&quot;</td>
<td>&quot;Numeric&quot;</td>
<td>&quot;Returns the absolute value of an integer.&quot;</td>
</tr>
<tr>
<td>&quot;abs&quot;</td>
<td>&quot;Numeric&quot;</td>
<td>&quot;Returns the absolute value of a floating point number.&quot;</td>
</tr>
<tr>
<td>&quot;acos&quot;</td>
<td>&quot;Trigonometric&quot;</td>
<td>&quot;Returns the arccosine of a number in radians.&quot;</td>
</tr>
<tr>
<td>&quot;all&quot;</td>
<td>&quot;Predicate&quot;</td>
<td>&quot;Returns true if the predicate holds for all elements in the given list.&quot;</td>
</tr>
<tr>
<td>&quot;any&quot;</td>
<td>&quot;Predicate&quot;</td>
<td>&quot;Returns true if the predicate holds for at least one element in the given list.&quot;</td>
</tr>
<tr>
<td>&quot;asin&quot;</td>
<td>&quot;Trigonometric&quot;</td>
<td>&quot;Returns the arcsine of a number in radians.&quot;</td>
</tr>
<tr>
<td>&quot;atan&quot;</td>
<td>&quot;Trigonometric&quot;</td>
<td>&quot;Returns the arctangent of a number in radians.&quot;</td>
</tr>
<tr>
<td>&quot;atan2&quot;</td>
<td>&quot;Trigonometric&quot;</td>
<td>&quot;Returns the arctangent2 of a set of coordinates in radians.&quot;</td>
</tr>
<tr>
<td>&quot;avg&quot;</td>
<td>&quot;Aggregating&quot;</td>
<td>&quot;Returns the average of a set of integer values.&quot;</td>
</tr>
<tr>
<td>&quot;avg&quot;</td>
<td>&quot;Aggregating&quot;</td>
<td>&quot;Returns the average of a set of floating point values.&quot;</td>
</tr>
</tbody>
</table>

Rows: 10

9.22. SHOW PROCEDURES

This section explains the SHOW PROCEDURES command.

Listing the available procedures can be done with SHOW PROCEDURES.

The command SHOW PROCEDURES returns only the default output. For a full output use the optional YIELD command. Full output: SHOW PROCEDURES YIELD *.

This command will produce a table with the following columns:

Table 250. List procedures output

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name of the procedure. [Default output]</td>
</tr>
<tr>
<td>description</td>
<td>The procedure description. [Default output]</td>
</tr>
<tr>
<td>mode</td>
<td>The procedure mode, for example READ or WRITE. [Default output]</td>
</tr>
<tr>
<td>Column</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>worksOnSystem</td>
<td>Whether the procedure can be run on the system database or not. Default output</td>
</tr>
<tr>
<td>signature</td>
<td>The signature of the procedure.</td>
</tr>
<tr>
<td>argumentDescription</td>
<td>List of the arguments for the procedure, as map of strings with name, type, default, and description.</td>
</tr>
<tr>
<td>returnDescription</td>
<td>List of the returned values for the procedure, as map of strings with name, type, and description.</td>
</tr>
<tr>
<td>admin</td>
<td>true if this procedure is an admin procedure.</td>
</tr>
<tr>
<td>rolesExecution</td>
<td>List of roles permitted to execute this procedure. Is null without the SHOW ROLE privilege.</td>
</tr>
<tr>
<td>rolesBoostedExecution</td>
<td>List of roles permitted to use boosted mode when executing this procedure. Is null without the SHOW ROLE privilege.</td>
</tr>
<tr>
<td>option</td>
<td>Map of extra output, e.g. if the procedure is deprecated.</td>
</tr>
</tbody>
</table>

### 9.22.1. Syntax

#### List all procedures

```
SHOW PROCEDURE[S]
[YIELD ( * | field[, ...] ) [ORDER BY field[, ...]] [SKIP n] [LIMIT n]]
[WHERE expression]
[RETURN field[, ...] [ORDER BY field[, ...]] [SKIP n] [LIMIT n]]
```

> When using the RETURN clause, the YIELD clause is mandatory and must not be omitted.

#### List procedures that the current user can execute

```
SHOW PROCEDURE[S] EXECUTABLE [BY CURRENT USER]
[YIELD ( * | field[, ...] ) [ORDER BY field[, ...]] [SKIP n] [LIMIT n]]
[WHERE expression]
[RETURN field[, ...] [ORDER BY field[, ...]] [SKIP n] [LIMIT n]]
```

> When using the RETURN clause, the YIELD clause is mandatory and must not be omitted.

#### List procedures that the specified user can execute
SHOW PROCEDURE[s] EXECUTABLE BY username
[YIELD { * | field[, ...] } [ORDER BY field[, ...]] [SKIP n] [LIMIT n]]
[WHERE expression]
[RETURN field[, ...]] [ORDER BY field[, ...]] [SKIP n] [LIMIT n]

Requires the privilege SHOW USER. This command cannot be used for LDAP users.

When using the RETURN clause, the YIELD clause is mandatory and must not be omitted.

9.22.2. Listing all procedures

To list all available procedures with the default output columns, the SHOW PROCEDURES command can be used. If all columns are required, use SHOW PROCEDURES YIELD *.

Query

SHOW PROCEDURES

Table 251. Result

<table>
<thead>
<tr>
<th>name</th>
<th>description</th>
<th>mode</th>
<th>worksOnSystem</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>db.awaitIndex</em></td>
<td>&quot;Wait for an index to come online (for example: CALL db.awaitIndex(&quot;MyIndex&quot;, 300)).&quot;</td>
<td>&quot;READ&quot;</td>
<td>true</td>
</tr>
<tr>
<td><em>db.awaitIndexes</em></td>
<td>&quot;Wait for all indexes to come online (for example: CALL db.awaitIndexes(300)).&quot;</td>
<td>&quot;READ&quot;</td>
<td>true</td>
</tr>
<tr>
<td><em>db.checkpoint</em></td>
<td>&quot;Initiate and wait for a new check point, or wait any already on-going check point to complete. Note that this temporarily disables the 'dbms.checkpoint.iops.limit' setting in order to make the check point complete faster. This might cause transaction throughput to degrade slightly, due to increased IO load.&quot;</td>
<td>&quot;DBMS&quot;</td>
<td>true</td>
</tr>
<tr>
<td><em>db.clearQueryCaches</em></td>
<td>&quot;Clears all query caches.&quot;</td>
<td>&quot;DBMS&quot;</td>
<td>true</td>
</tr>
<tr>
<td><em>db.constraints</em></td>
<td>&quot;List all constraints in the database.&quot;</td>
<td>&quot;READ&quot;</td>
<td>true</td>
</tr>
<tr>
<td><em>db.createIndex</em></td>
<td>&quot;Create a named schema index with specified index provider and configuration (optional). Yield: name, labels, properties, providerName, status&quot;</td>
<td>&quot;SCHEMA&quot;</td>
<td>false</td>
</tr>
<tr>
<td><em>db.createLabel</em></td>
<td>&quot;Create a label&quot;</td>
<td>&quot;WRITE&quot;</td>
<td>false</td>
</tr>
<tr>
<td>name</td>
<td>description</td>
<td>mode</td>
<td>worksOnSystem</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>------------</td>
<td>---------------</td>
</tr>
<tr>
<td>&quot;db.createNodeKey&quot;</td>
<td>&quot;Create a named node key constraint. Backing index will use specified index provider and configuration (optional). Yield: name, labels, properties, providerName, status&quot;</td>
<td>&quot;SCHEMA&quot;</td>
<td>false</td>
</tr>
<tr>
<td>&quot;db.createProperty&quot;</td>
<td>&quot;Create a Property&quot;</td>
<td>&quot;WRITE&quot;</td>
<td>false</td>
</tr>
<tr>
<td>&quot;db.createRelationshipType&quot;</td>
<td>&quot;Create a RelationshipType&quot;</td>
<td>&quot;WRITE&quot;</td>
<td>false</td>
</tr>
<tr>
<td>&quot;db.createUniquePropertyConstraint&quot;</td>
<td>&quot;Create a named unique property constraint. Backing index will use specified index provider and configuration (optional). Yield: name, labels, properties, providerName, status&quot;</td>
<td>&quot;SCHEMA&quot;</td>
<td>false</td>
</tr>
<tr>
<td>&quot;db.index.fulltext.await EventuallyConsistentIndexRefresh&quot;</td>
<td>&quot;Wait for the updates from recently committed transactions to be applied to any eventually-consistent full-text indexes.&quot;</td>
<td>&quot;READ&quot;</td>
<td>true</td>
</tr>
<tr>
<td>&quot;db.index.fulltext.createNodeIndex&quot;</td>
<td>&quot;Create a node full-text index for the given labels and properties. The optional 'config' map parameter can be used to supply settings to the index. Supported settings are 'analyzer', for specifying what analyzer to use when indexing and querying. Use the 'db.index.fulltext.listAvailableAnalyzers' procedure to see what options are available. And 'eventually_consistent' which can be set to 'true' to make this index eventually consistent, such that updates from committing transactions are applied in a background thread.&quot;</td>
<td>&quot;SCHEMA&quot;</td>
<td>false</td>
</tr>
</tbody>
</table>
### 9.22.3. Listing procedures with filtering on output columns

The listed procedures can be filtered in multiple ways, one way is to use the `WHERE` clause. For example, returning the names of all admin procedures:

**Query**

```sql
SHOW PROCEDURES YIELD name, admin
WHERE admin
```

**Table 252. Result**

<table>
<thead>
<tr>
<th>name</th>
<th>admin</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;db.clearQueryCaches&quot;</td>
<td>true</td>
</tr>
<tr>
<td>&quot;db.listLocks&quot;</td>
<td>true</td>
</tr>
<tr>
<td>&quot;db.prepareForReplanning&quot;</td>
<td>true</td>
</tr>
<tr>
<td>&quot;db.stats.clear&quot;</td>
<td>true</td>
</tr>
<tr>
<td>&quot;db.stats.collect&quot;</td>
<td>true</td>
</tr>
<tr>
<td>&quot;db.stats.retrieve&quot;</td>
<td>true</td>
</tr>
<tr>
<td>&quot;db.stats.retrieveAllAnonymized&quot;</td>
<td>true</td>
</tr>
</tbody>
</table>

Rows: 7
9.22.4. Listing procedures with other filtering

The listed procedures can also be filtered by whether a user can execute them. This filtering is only available through the EXECUTABLE clause and not through the WHERE clause. This is due to using the user’s privileges instead of filtering on the available output columns.

There are two options, how to use the EXECUTABLE clause. The first option, is to filter for the current user:

**Query**

```
SHOW PROCEDURES EXECUTABLE BY CURRENT USER YIELD *
```

**Table 253. Result**

<table>
<thead>
<tr>
<th>name</th>
<th>description</th>
<th>rolesExecution</th>
<th>rolesBoostedExecution</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;db.awaitIndex&quot;</td>
<td>&quot;Wait for an index to come online (for example: CALL db.awaitIndex(&quot;MyIndex&quot;, 300)).&quot;</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;db.awaitIndexes&quot;</td>
<td>&quot;Wait for all indexes to come online (for example: CALL db.awaitIndexes(300)).&quot;</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;db.checkpoint&quot;</td>
<td>&quot;Initiate and wait for a new check point, or wait any already on-going check point to complete. Note that this temporarily disables the 'dbms.checkpoint.iops.limit' setting in order to make the check point complete faster. This might cause transaction throughput to degrade slightly, due to increased IO load.&quot;</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;db.constraints&quot;</td>
<td>&quot;List all constraints in the database.&quot;</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;db.createIndex&quot;</td>
<td>&quot;Create a named schema index with specified index provider and configuration (optional). Yield: name, labels, properties, providerName, status&quot;</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;db.createLabel&quot;</td>
<td>&quot;Create a label&quot;</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>name</td>
<td>description</td>
<td>rolesExecution</td>
<td>rolesBoostedExecution</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>&quot;db.createNodeKey&quot;</td>
<td>&quot;Create a named node key constraint. Backing index will use specified index provider and configuration (optional). Yield: name, labels, properties, providerName, status&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;db.createProperty&quot;</td>
<td>&quot;Create a Property&quot;</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;db.createRelationshipType&quot;</td>
<td>&quot;Create a RelationshipType&quot;</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;db.createUniquePropertyConstraint&quot;</td>
<td>&quot;Create a named unique property constraint. Backing index will use specified index provider and configuration (optional). Yield: name, labels, properties, providerName, status&quot;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Rows: 10

Note that the two roles columns are empty due to missing the `SHOW ROLE` privilege.

The second option, filters the list to only contain procedures executable by a specific user:

**Query**

```
SHOW PROCEDURES EXECUTABLE BY jake
```

**Table 254. Result**

<table>
<thead>
<tr>
<th>name</th>
<th>description</th>
<th>mode</th>
<th>worksOnSystem</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;db.awaitIndex&quot;</td>
<td>&quot;Wait for an index to come online (for example: CALL db.awaitIndex(&quot;MyIndex&quot;, 300)).&quot;</td>
<td>&quot;READ&quot;</td>
<td>true</td>
</tr>
<tr>
<td>&quot;db.awaitIndexes&quot;</td>
<td>&quot;Wait for all indexes to come online (for example: CALL db.awaitIndexes(300)).&quot;</td>
<td>&quot;READ&quot;</td>
<td>true</td>
</tr>
<tr>
<td>name</td>
<td>description</td>
<td>mode</td>
<td>worksOnSystem</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------</td>
<td>---------------</td>
</tr>
<tr>
<td>&quot;db.checkpoint&quot;</td>
<td>&quot;Initiate and wait for a new check point, or wait any already on-going check point to complete. Note that this temporarily disables the 'dbms.checkpoint.iops.limit' setting in order to make the check point complete faster. This might cause transaction throughput to degrade slightly, due to increased IO load.&quot;</td>
<td>&quot;DBMS&quot;</td>
<td>true</td>
</tr>
<tr>
<td>&quot;db.constraints&quot;</td>
<td>&quot;List all constraints in the database.&quot;</td>
<td>&quot;READ&quot;</td>
<td>true</td>
</tr>
<tr>
<td>&quot;db.createIndex&quot;</td>
<td>&quot;Create a named schema index with specified index provider and configuration (optional). Yield: name, labels, properties, providerName, status&quot;</td>
<td>&quot;SCHEMA&quot;</td>
<td>false</td>
</tr>
<tr>
<td>&quot;db.createLabel&quot;</td>
<td>&quot;Create a label&quot;</td>
<td>&quot;WRITE&quot;</td>
<td>false</td>
</tr>
<tr>
<td>&quot;db.createNodeKey&quot;</td>
<td>&quot;Create a named node key constraint. Backing index will use specified index provider and configuration (optional). Yield: name, labels, properties, providerName, status&quot;</td>
<td>&quot;SCHEMA&quot;</td>
<td>false</td>
</tr>
<tr>
<td>&quot;db.createProperty&quot;</td>
<td>&quot;Create a Property&quot;</td>
<td>&quot;WRITE&quot;</td>
<td>false</td>
</tr>
<tr>
<td>&quot;db.createRelationshipType&quot;</td>
<td>&quot;Create a RelationshipType&quot;</td>
<td>&quot;WRITE&quot;</td>
<td>false</td>
</tr>
<tr>
<td>&quot;db.createUniquePropertyConstraint&quot;</td>
<td>&quot;Create a named unique property constraint. Backing index will use specified index provider and configuration (optional). Yield: name, labels, properties, providerName, status&quot;</td>
<td>&quot;SCHEMA&quot;</td>
<td>false</td>
</tr>
</tbody>
</table>

Rows: 10

Chapter 10. 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 - instant types [Summary][Detail]
- Temporal functions - duration [Summary][Detail]
- Spatial functions [Summary][Detail]
- LOAD CSV functions [Summary][Detail]
- User-defined functions [Summary][Detail]

Related information may be found in Operators.

Please note
- Functions in Cypher 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, the size() function applied to any Unicode character will return 1, even if the character does not fit in the 16 bits of one char.

Example 5. List available functions

To list the available functions, run the following Cypher query:

```
SHOW FUNCTIONS
```

Predicate functions

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>all()</td>
<td>all(variable :: VARIABLE IN list :: LIST OF ANY? WHERE predicate :: ANY?) :: (BOOLEAN?)</td>
<td>Returns true if the predicate holds for all elements in the given list.</td>
</tr>
</tbody>
</table>
## Function Signature Description

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>any()</td>
<td>any(variable :: VARIABLE IN list :: LIST OF ANY? WHERE predicate :: ANY?) :: (BOOLEAN?)</td>
<td>Returns true if the predicate holds for at least one element in the given list.</td>
</tr>
<tr>
<td>exists()</td>
<td>exists(input :: ANY?) :: (BOOLEAN?)</td>
<td>Returns true if a match for the pattern exists in the graph, or if the specified property exists in the node, relationship or map.</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>isEmpty(input :: LIST? OF ANY?) :: (BOOLEAN?)</td>
<td>Checks whether a list is empty.</td>
</tr>
<tr>
<td></td>
<td>isEmpty(input :: MAP?) :: (BOOLEAN?)</td>
<td>Checks whether a map is empty.</td>
</tr>
<tr>
<td></td>
<td>isEmpty(input :: STRING?) :: (BOOLEAN?)</td>
<td>Checks whether a string is empty.</td>
</tr>
<tr>
<td>none()</td>
<td>none(variable :: VARIABLE IN list :: LIST OF ANY? WHERE predicate :: ANY?) :: (BOOLEAN?)</td>
<td>Returns true if the predicate holds for no element in the given list.</td>
</tr>
<tr>
<td>single()</td>
<td>single(variable :: VARIABLE IN list :: LIST OF ANY? WHERE predicate :: ANY?) :: (BOOLEAN?)</td>
<td>Returns true if the predicate holds for exactly one of the elements in the given list.</td>
</tr>
</tbody>
</table>

### Scalar functions

These functions return a single value.

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>coalesce()</td>
<td>coalesce(input :: ANY?) :: (ANY?)</td>
<td>Returns the first non-null value in a list of expressions.</td>
</tr>
<tr>
<td>endNode()</td>
<td>endNode(input :: RELATIONSHIP?) :: (NODE?)</td>
<td>Returns the end node of a relationship.</td>
</tr>
<tr>
<td>head()</td>
<td>head(list :: LIST? OF ANY?) :: (ANY?)</td>
<td>Returns the first element in a list.</td>
</tr>
<tr>
<td>id()</td>
<td>id(input :: NODE?) :: (INTEGER?)</td>
<td>Returns the id of a node.</td>
</tr>
<tr>
<td></td>
<td>id(input :: RELATIONSHIP?) :: (INTEGER?)</td>
<td>Returns the id of a relationship.</td>
</tr>
<tr>
<td>last()</td>
<td>last(list :: LIST? OF ANY?) :: (ANY?)</td>
<td>Returns the last element in a list.</td>
</tr>
<tr>
<td>length()</td>
<td>length(input :: PATH?) :: (INTEGER?)</td>
<td>Returns the length of a path.</td>
</tr>
<tr>
<td>properties()</td>
<td>properties(input :: MAP?) :: (MAP?)</td>
<td>Returns a map containing all the properties of a map.</td>
</tr>
<tr>
<td></td>
<td>properties(input :: NODE?) :: (MAP?)</td>
<td>Returns a map containing all the properties of a node.</td>
</tr>
<tr>
<td></td>
<td>properties(input :: RELATIONSHIP?) :: (MAP?)</td>
<td>Returns a map containing all the properties of a relationship.</td>
</tr>
<tr>
<td>randomUUID()</td>
<td>randomUUID() :: (STRING?)</td>
<td>Generates a random UUID.</td>
</tr>
<tr>
<td>Function</td>
<td>Signature</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>size()</td>
<td>size(input :: LIST? OF ANY?) :: (INTEGER?)</td>
<td>Returns the number of items in a list.</td>
</tr>
<tr>
<td></td>
<td>size(input :: STRING?) :: (INTEGER?)</td>
<td>Returns the number of Unicode characters in a string.</td>
</tr>
<tr>
<td>startNode()</td>
<td>startNode(input :: RELATIONSHIP?) :: (NODE?)</td>
<td>Returns the start node of a relationship.</td>
</tr>
<tr>
<td>toBoolean()</td>
<td>toBoolean(input :: STRING?) :: (BOOLEAN?)</td>
<td>Converts a string value to a boolean value.</td>
</tr>
<tr>
<td></td>
<td>toBoolean(input :: BOOLEAN?) :: (BOOLEAN?)</td>
<td>Converts a boolean value to a boolean value.</td>
</tr>
<tr>
<td></td>
<td>toBoolean(input :: INTEGER?) :: (BOOLEAN?)</td>
<td>Converts an integer value to a boolean value.</td>
</tr>
<tr>
<td>toBooleanOrNull()</td>
<td>toBooleanOrNull(input :: ANY?) :: (BOOLEAN?)</td>
<td>Converts a value to a boolean value, or null if the value cannot be converted.</td>
</tr>
<tr>
<td>toFloat()</td>
<td>toFloat(input :: NUMBER?) :: (FLOAT?)</td>
<td>Converts a number value to a floating point value.</td>
</tr>
<tr>
<td></td>
<td>toFloat(input :: STRING?) :: (FLOAT?)</td>
<td>Converts a string value to a floating point value.</td>
</tr>
<tr>
<td>toFloatOrNull()</td>
<td>toFloatOrNull(input :: ANY?) :: (FLOAT?)</td>
<td>Converts a value to a floating point value, or null if the value cannot be converted.</td>
</tr>
<tr>
<td>toInteger()</td>
<td>toInteger(input :: NUMBER?) :: (INTEGER?)</td>
<td>Converts a number value to an integer value.</td>
</tr>
<tr>
<td></td>
<td>toInteger(input :: BOOLEAN?) :: (INTEGER?)</td>
<td>Converts a boolean value to an integer value.</td>
</tr>
<tr>
<td></td>
<td>toInteger(input :: STRING?) :: (INTEGER?)</td>
<td>Converts a string value to an integer value.</td>
</tr>
<tr>
<td>toIntegerOrNull()</td>
<td>toIntegerOrNull(input :: ANY?) :: (INTEGER?)</td>
<td>Converts a value to an integer value, or null if the value cannot be converted.</td>
</tr>
<tr>
<td>type()</td>
<td>type(input :: RELATIONSHIP?) :: (STRING?)</td>
<td>Returns the string representation of the relationship type.</td>
</tr>
</tbody>
</table>

**Aggregating functions**

These functions take multiple values as arguments, and calculate and return an aggregated value from them.
<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg()</td>
<td>avg(input :: DURATION?) :: (DURATION?)</td>
<td>Returns the average of a set of duration values.</td>
</tr>
<tr>
<td></td>
<td>avg(input :: FLOAT?) :: (FLOAT?)</td>
<td>Returns the average of a set of floating point values.</td>
</tr>
<tr>
<td></td>
<td>avg(input :: INTEGER?) :: (INTEGER?)</td>
<td>Returns the average of a set of integer values.</td>
</tr>
<tr>
<td>collect()</td>
<td>collect(input :: ANY?) :: (LIST? OF ANY?)</td>
<td>Returns a list containing the values returned by an expression.</td>
</tr>
<tr>
<td>count()</td>
<td>count(input :: ANY?) :: (INTEGER?)</td>
<td>Returns the number of values or rows.</td>
</tr>
<tr>
<td>max()</td>
<td>max(input :: ANY?) :: (ANY?)</td>
<td>Returns the maximum value in a set of values.</td>
</tr>
<tr>
<td>min()</td>
<td>min(input :: ANY?) :: (ANY?)</td>
<td>Returns the minimum value in a set of values.</td>
</tr>
<tr>
<td>percentileCont()</td>
<td>percentileCont(input :: FLOAT?, percentile :: FLOAT?) :: (FLOAT?)</td>
<td>Returns the percentile of a value over a group using linear interpolation.</td>
</tr>
<tr>
<td>percentileDisc()</td>
<td>percentileDisc(input :: FLOAT?, percentile :: FLOAT?) :: (FLOAT?)</td>
<td>Returns the nearest floating point value to the given percentile over a group using a rounding method.</td>
</tr>
<tr>
<td></td>
<td>percentileDisc(input :: INTEGER?, percentile :: FLOAT?) :: (INTEGER?)</td>
<td>Returns the nearest integer value to the given percentile over a group using a rounding method.</td>
</tr>
<tr>
<td>stdev()</td>
<td>stdev(input :: FLOAT?) :: (FLOAT?)</td>
<td>Returns the standard deviation for the given value over a group for a sample of a population.</td>
</tr>
<tr>
<td>stdevp()</td>
<td>stdevp(input :: FLOAT?) :: (FLOAT?)</td>
<td>Returns the standard deviation for the given value over a group for an entire population.</td>
</tr>
<tr>
<td>sum()</td>
<td>sum(input :: DURATION?) :: (DURATION?)</td>
<td>Returns the sum of a set of durations</td>
</tr>
<tr>
<td></td>
<td>sum(input :: FLOAT?) :: (FLOAT?)</td>
<td>Returns the sum of a set of floats</td>
</tr>
<tr>
<td></td>
<td>sum(input :: INTEGER?) :: (INTEGER?)</td>
<td>Returns the sum of a set of integers</td>
</tr>
</tbody>
</table>

### List functions

These functions return lists of other values. Further details and examples of lists may be found in Lists.
<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>keys()</td>
<td>keys(input :: MAP?) :: (LIST? OF STRING?)</td>
<td>Returns a list containing the string representations for all the property names of a map.</td>
</tr>
<tr>
<td></td>
<td>keys(input :: NODE?) :: (LIST? OF STRING?)</td>
<td>Returns a list containing the string representations for all the property names of a node.</td>
</tr>
<tr>
<td></td>
<td>keys(input :: RELATIONSHIP?) :: (LIST? OF STRING?)</td>
<td>Returns a list containing the string representations for all the property names of a relationship</td>
</tr>
<tr>
<td>labels()</td>
<td>labels(input :: NODE?) :: (LIST? OF STRING?)</td>
<td>Returns a list containing the string representations for all the labels of a node.</td>
</tr>
<tr>
<td>nodes()</td>
<td>nodes(input :: PATH?) :: (LIST? OF NODE?)</td>
<td>Returns a list containing all the nodes in a path.</td>
</tr>
<tr>
<td>range()</td>
<td>range(start :: INTEGER?, end :: INTEGER?) :: (LIST? OF INTEGER?)</td>
<td>Returns a list comprising all integer values within a specified range.</td>
</tr>
<tr>
<td></td>
<td>range(start :: INTEGER?, end :: INTEGER?, step :: INTEGER?) :: (LIST? OF INTEGER?)</td>
<td>Returns a list comprising all integer values within a specified range created with step length.</td>
</tr>
<tr>
<td>reduce()</td>
<td>reduce(accumulator :: VARIABLE = initial :: ANY?, variable :: VARIABLE IN list :: LIST OF ANY?</td>
<td>Runs an expression against individual elements of a list, storing the result of the expression in an accumulator.</td>
</tr>
<tr>
<td></td>
<td>expression :: ANY) :: (ANY?)</td>
<td></td>
</tr>
<tr>
<td>relationships()</td>
<td>relationships(input :: PATH?) :: (LIST? OF RELATIONSHIP?)</td>
<td>Returns a list containing all the relationships in a path.</td>
</tr>
<tr>
<td>reverse()</td>
<td>reverse(input :: LIST? OF ANY?) :: (LIST? OF ANY?)</td>
<td>Returns a list in which the order of all elements in the original list have been reversed.</td>
</tr>
<tr>
<td>tail()</td>
<td>tail(input :: LIST? OF ANY?) :: (LIST? OF ANY?)</td>
<td>Returns all but the first element in a list.</td>
</tr>
<tr>
<td>toBooleanList()</td>
<td>toBooleanList(input :: LIST? OF ANY?) :: (LIST? OF BOOLEAN?)</td>
<td>Converts a list of values to a list of boolean values. If any values are not convertible to boolean they will be null in the list returned.</td>
</tr>
<tr>
<td>toFloatList()</td>
<td>toFloatList(input :: LIST? OF ANY?) :: (LIST? OF FLOAT?)</td>
<td>Converts a list of values to a list of floating point values. If any values are not convertible to floating point they will be null in the list returned.</td>
</tr>
<tr>
<td>toIntegerList()</td>
<td>toIntegerList(input :: LIST? OF ANY?) :: (LIST? OF INTEGER?)</td>
<td>Converts a list of values to a list of integer values. If any values are not convertible to integer they will be null in the list returned.</td>
</tr>
</tbody>
</table>
### Function

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>toStringList()</td>
<td><code>toStringList(input :: LIST? OF ANY?) :: (LIST? OF STRING?)</code></td>
<td>Converts a list of values to a list of string values. If any values are not convertible to string they will be null in the list returned.</td>
</tr>
</tbody>
</table>

#### Numeric functions

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs()</td>
<td><code>abs(input :: FLOAT?) :: (FLOAT?)</code></td>
<td>Returns the absolute value of a floating point number.</td>
</tr>
<tr>
<td></td>
<td><code>abs(input :: INTEGER?) :: (INTEGER?)</code></td>
<td>Returns the absolute value of an integer.</td>
</tr>
<tr>
<td>ceil()</td>
<td><code>ceil(input :: FLOAT?) :: (FLOAT?)</code></td>
<td>Returns the smallest floating point number that is greater than or equal to a number and equal to a mathematical integer.</td>
</tr>
<tr>
<td>floor()</td>
<td><code>floor(input :: FLOAT?) :: (FLOAT?)</code></td>
<td>Returns the largest floating point number that is less than or equal to a number and equal to a mathematical integer.</td>
</tr>
<tr>
<td>rand()</td>
<td><code>rand() :: (FLOAT?)</code></td>
<td>Returns a random floating point number in the range from 0 (inclusive) to 1 (exclusive); i.e. [0,1).</td>
</tr>
<tr>
<td>round()</td>
<td><code>round(input :: FLOAT?) :: (FLOAT?)</code></td>
<td>Returns the value of a number rounded to the nearest integer.</td>
</tr>
<tr>
<td></td>
<td><code>round(value :: FLOAT?, precision :: NUMBER?) :: (FLOAT?)</code></td>
<td>Returns the value of a number rounded to the specified precision using rounding mode HALF_UP.</td>
</tr>
<tr>
<td></td>
<td><code>round(value :: FLOAT?, precision :: NUMBER?, mode :: STRING?) :: (FLOAT?)</code></td>
<td>Returns the value of a number rounded to the specified precision with the specified rounding mode.</td>
</tr>
<tr>
<td>sign()</td>
<td><code>sign(input :: FLOAT?) :: (INTEGER?)</code></td>
<td>Returns the signum of a floating point number: 0 if the number is 0, -1 for any negative number, and 1 for any positive number.</td>
</tr>
<tr>
<td></td>
<td><code>sign(input :: INTEGER?) :: (INTEGER?)</code></td>
<td>Returns the signum of an integer number: 0 if the number is 0, -1 for any negative number, and 1 for any positive number.</td>
</tr>
</tbody>
</table>

#### Logarithmic functions

These functions all operate on numerical expressions only, and will return an error if used on any other values.
<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>e()</td>
<td>e() :: (FLOAT?)</td>
<td>Returns the base of the natural logarithm, e.</td>
</tr>
<tr>
<td>exp()</td>
<td>exp(input :: FLOAT?) :: (FLOAT?)</td>
<td>Returns (e^n), where (e) is the base of the natural logarithm, and (n) is the value of the argument expression.</td>
</tr>
<tr>
<td>log()</td>
<td>log(input :: FLOAT?) :: (FLOAT?)</td>
<td>Returns the natural logarithm of a number.</td>
</tr>
<tr>
<td>log10()</td>
<td>log10(input :: FLOAT?) :: (FLOAT?)</td>
<td>Returns the common logarithm (base 10) of a number.</td>
</tr>
<tr>
<td>sqrt()</td>
<td>sqrt(input :: FLOAT?) :: (FLOAT?)</td>
<td>Returns the square root of a number.</td>
</tr>
</tbody>
</table>

**Trigonometric functions**

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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acos()</td>
<td>acos(input :: FLOAT?) :: (FLOAT?)</td>
<td>Returns the arccosine of a number in radians.</td>
</tr>
<tr>
<td>asin()</td>
<td>asin(input :: FLOAT?) :: (FLOAT?)</td>
<td>Returns the arcsine of a number in radians.</td>
</tr>
<tr>
<td>atan()</td>
<td>atan(input :: FLOAT?) :: (FLOAT?)</td>
<td>Returns the arctangent of a number in radians.</td>
</tr>
<tr>
<td>atan2()</td>
<td>atan2(y :: FLOAT?, x :: FLOAT?) :: (FLOAT?)</td>
<td>Returns the arctangent2 of a set of coordinates in radians.</td>
</tr>
<tr>
<td>cos()</td>
<td>cos(input :: FLOAT?) :: (FLOAT?)</td>
<td>Returns the cosine of a number.</td>
</tr>
<tr>
<td>cot()</td>
<td>cot(input :: FLOAT?) :: (FLOAT?)</td>
<td>Returns the cotangent of a number.</td>
</tr>
<tr>
<td>degrees()</td>
<td>degrees(input :: FLOAT?) :: (FLOAT?)</td>
<td>Converts radians to degrees.</td>
</tr>
<tr>
<td>haversin()</td>
<td>haversin(input :: FLOAT?) :: (FLOAT?)</td>
<td>Returns half the versine of a number.</td>
</tr>
<tr>
<td>pi()</td>
<td>pi() :: (FLOAT?)</td>
<td>Returns the mathematical constant pi.</td>
</tr>
<tr>
<td>radians()</td>
<td>radians(input :: FLOAT?) :: (FLOAT?)</td>
<td>Converts degrees to radians.</td>
</tr>
<tr>
<td>sin()</td>
<td>sin(input :: FLOAT?) :: (FLOAT?)</td>
<td>Returns the sine of a number.</td>
</tr>
<tr>
<td>tan()</td>
<td>tan(input :: FLOAT?) :: (FLOAT?)</td>
<td>Returns the tangent of a number.</td>
</tr>
</tbody>
</table>

**String functions**

These functions are used to manipulate strings or to create a string representation of another value.
<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>left()</td>
<td>left(original :: STRING?, length :: INTEGER?) :: (STRING?)</td>
<td>Returns a string containing the specified number of leftmost characters of the original string.</td>
</tr>
<tr>
<td>ltrim()</td>
<td>ltrim(input :: STRING?) :: (STRING?)</td>
<td>Returns the original string with leading whitespace removed.</td>
</tr>
<tr>
<td>replace()</td>
<td>replace(original :: STRING?, search :: STRING?, replace :: STRING?) :: (STRING?)</td>
<td>Returns a string in which all occurrences of a specified search string in the original string have been replaced by another (specified) replace string.</td>
</tr>
<tr>
<td>reverse()</td>
<td>reverse(input :: STRING?) :: (STRING?)</td>
<td>Returns a string in which the order of all characters in the original string have been reversed.</td>
</tr>
<tr>
<td>right()</td>
<td>right(original :: STRING?, length :: INTEGER?) :: (STRING?)</td>
<td>Returns a string containing the specified number of rightmost characters of the original string.</td>
</tr>
<tr>
<td>rtrim()</td>
<td>rtrim(input :: STRING?) :: (STRING?)</td>
<td>Returns the original string with trailing whitespace removed.</td>
</tr>
<tr>
<td>split()</td>
<td>split(original :: STRING?, splitDelimiter :: STRING?) :: (LIST? OF STRING?)</td>
<td>Returns a list of strings resulting from the splitting of the original string around matches of the given delimiter.</td>
</tr>
<tr>
<td></td>
<td>split(original :: STRING?, splitDelimiters :: LIST? OF STRING?) :: (LIST? OF STRING?)</td>
<td>Returns a list of strings resulting from the splitting of the original string around matches of any of the given delimiters.</td>
</tr>
<tr>
<td>substring()</td>
<td>substring(original :: STRING?, start :: INTEGER?) :: (STRING?)</td>
<td>Returns a substring of the original string, beginning with a 0-based index start.</td>
</tr>
<tr>
<td></td>
<td>substring(original :: STRING?, start :: INTEGER?, length :: INTEGER?) :: (STRING?)</td>
<td>Returns a substring of length 'length' of the original string, beginning with a 0-based index start.</td>
</tr>
<tr>
<td>toLower()</td>
<td>toLower(input :: STRING?) :: (STRING?)</td>
<td>Returns the original string in lowercase.</td>
</tr>
<tr>
<td>toString()</td>
<td>toString(input :: ANY?) :: (STRING?)</td>
<td>Converts an integer, float, boolean, point or temporal type (i.e. Date, Time, LocalTime, DateTime, LocalDateTime or Duration) value to a string.</td>
</tr>
<tr>
<td>toStringOrNull()</td>
<td>toStringOrNull(input :: ANY?) :: (STRING?)</td>
<td>Converts an integer, float, boolean, point or temporal type (i.e. Date, Time, LocalTime, DateTime, LocalDateTime or Duration) value to a string, or null if the value cannot be converted.</td>
</tr>
<tr>
<td>toUpper()</td>
<td>toUpper(input :: STRING?) :: (STRING?)</td>
<td>Returns the original string in uppercase.</td>
</tr>
<tr>
<td>trim()</td>
<td>trim(input :: STRING?) :: (STRING?)</td>
<td>Returns the original string with leading and trailing whitespace removed.</td>
</tr>
</tbody>
</table>
**Temporal instant types functions**

Values of the **temporal types** — Date, Time, LocalTime, DateTime, and LocalDateTime — can be created manipulated using the following functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date()</td>
<td><code>date(input = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (DATE?)</code></td>
<td>Create a Date instant.</td>
</tr>
<tr>
<td>date.realtime()</td>
<td><code>date.realtime(timezone = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (DATE?)</code></td>
<td>Get the current Date instant using the realtime clock.</td>
</tr>
<tr>
<td>date.statement()</td>
<td><code>date.statement(timezone = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (DATE?)</code></td>
<td>Get the current Date instant using the statement clock.</td>
</tr>
<tr>
<td>date.transaction()</td>
<td><code>date.transaction(timezone = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (DATE?)</code></td>
<td>Get the current Date instant using the transaction clock.</td>
</tr>
<tr>
<td>date.truncate()</td>
<td><code>date.truncate(unit :: STRING?, input = DEFAULT_TEMPORAL_ARGUMENT :: ANY?, fields = null :: MAP?) :: (DATE?)</code></td>
<td>Truncate the input temporal value to a Date instant using the specified unit.</td>
</tr>
<tr>
<td>datetime()</td>
<td><code>datetime(input = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (DATETIME?)</code></td>
<td>Create a DateTime instant.</td>
</tr>
<tr>
<td>datetime.fromepoch()</td>
<td><code>datetime.fromepoch(seconds :: NUMBER?, nanoseconds :: NUMBER?) :: (DATETIME?)</code></td>
<td>Create a DateTime given the seconds and nanoseconds since the start of the epoch.</td>
</tr>
<tr>
<td>datetime.fromepochmillis()</td>
<td><code>datetime.fromepochmillis(millisecond s :: NUMBER?) :: (DATETIME?)</code></td>
<td>Create a DateTime given the milliseconds since the start of the epoch.</td>
</tr>
<tr>
<td>datetime.realtime()</td>
<td><code>datetime.realtime(timezone = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (DATETIME?)</code></td>
<td>Get the current DateTime instant using the realtime clock.</td>
</tr>
<tr>
<td>datetime.statement()</td>
<td><code>datetime.statement(timezone = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (DATETIME?)</code></td>
<td>Get the current DateTime instant using the statement clock.</td>
</tr>
<tr>
<td>datetime.transaction()</td>
<td><code>datetime.transaction(timezone = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (DATETIME?)</code></td>
<td>Get the current DateTime instant using the transaction clock.</td>
</tr>
<tr>
<td>datetime.truncate()</td>
<td><code>datetime.truncate(unit :: STRING?, input = DEFAULT_TEMPORAL_ARGUMENT :: ANY?, fields = null :: MAP?) :: (DATETIME?)</code></td>
<td>Truncate the input temporal value to a DateTime instant using the specified unit.</td>
</tr>
<tr>
<td>localdatetime()</td>
<td><code>localdatetime(input = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (LOCALDATETIME?)</code></td>
<td>Create a LocalDateTime instant.</td>
</tr>
<tr>
<td>localdatetime.realtime()</td>
<td><code>localdatetime.realtime(timezone = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (LOCALDATETIME?)</code></td>
<td>Get the current LocalDateTime instant using the realtime clock.</td>
</tr>
<tr>
<td>localdatetime.statement()</td>
<td><code>localdatetime.statement(timezone = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (LOCALDATETIME?)</code></td>
<td>Get the current LocalDateTime instant using the statement clock.</td>
</tr>
<tr>
<td>Function</td>
<td>Signature</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>---------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>localdatetime.transaction()</code></td>
<td><code>localdatetime.transaction(timezone = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (LOCALDATETIME?)</code></td>
<td>Get the current LocalDateTime instant using the transaction clock.</td>
</tr>
<tr>
<td><code>localdatetime.truncate()</code></td>
<td><code>localdatetime.truncate(unit :: STRING?, input = DEFAULT_TEMPORAL_ARGUMENT :: ANY?, fields = null :: MAP?) :: (LOCALDATETIME?)</code></td>
<td>Truncate the input temporal value to a LocalDateTime instant using the specified unit.</td>
</tr>
<tr>
<td><code>localtime()</code></td>
<td><code>localtime(input = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (LOCALTIME?)</code></td>
<td>Create a LocalTime instant.</td>
</tr>
<tr>
<td><code>localtime.realtime()</code></td>
<td><code>localtime.realtime(timezone = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (LOCALTIME?)</code></td>
<td>Get the current LocalTime instant using the realtime clock.</td>
</tr>
<tr>
<td><code>localtime.statement()</code></td>
<td><code>localtime.statement(timezone = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (LOCALTIME?)</code></td>
<td>Get the current LocalTime instant using the statement clock.</td>
</tr>
<tr>
<td><code>localtime.transaction()</code></td>
<td><code>localtime.transaction(timezone = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (LOCALTIME?)</code></td>
<td>Get the current LocalTime instant using the transaction clock.</td>
</tr>
<tr>
<td><code>localtime.truncate()</code></td>
<td><code>localtime.truncate(unit :: STRING?, input = DEFAULT_TEMPORAL_ARGUMENT :: ANY?, fields = null :: MAP?) :: (LOCALTIME?)</code></td>
<td>Truncate the input temporal value to a LocalTime instant using the specified unit.</td>
</tr>
<tr>
<td><code>time()</code></td>
<td><code>time(input = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (TIME?)</code></td>
<td>Create a Time instant.</td>
</tr>
<tr>
<td><code>time.realtime()</code></td>
<td><code>time.realtime(timezone = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (TIME?)</code></td>
<td>Get the current Time instant using the realtime clock.</td>
</tr>
<tr>
<td><code>time.statement()</code></td>
<td><code>time.statement(timezone = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (TIME?)</code></td>
<td>Get the current Time instant using the statement clock.</td>
</tr>
<tr>
<td><code>time.transaction()</code></td>
<td><code>time.transaction(timezone = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (TIME?)</code></td>
<td>Get the current Time instant using the transaction clock.</td>
</tr>
<tr>
<td><code>time.truncate()</code></td>
<td><code>time.truncate(unit :: STRING?, input = DEFAULT_TEMPORAL_ARGUMENT :: ANY?, fields = null :: MAP?) :: (TIME?)</code></td>
<td>Truncate the input temporal value to a Time instant using the specified unit.</td>
</tr>
</tbody>
</table>

**Temporal duration functions**

Duration values of the **temporal types** can be created manipulated using the following functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>duration()</code></td>
<td><code>duration(input :: ANY?) :: (DURATION?)</code></td>
<td>Construct a Duration value.</td>
</tr>
<tr>
<td><code>duration.between()</code></td>
<td><code>duration.between(from :: ANY?, to :: ANY?) :: (DURATION?)</code></td>
<td>Compute the duration between the 'from' instant (inclusive) and the 'to' instant (exclusive) in logical units.</td>
</tr>
</tbody>
</table>
### Function

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>duration.inDays()</td>
<td>duration.inDays(from :: ANY?, to :: ANY?) :: (DURATION?)</td>
<td>Compute the duration between the ‘from’ instant (inclusive) and the ‘to’ instant (exclusive) in days.</td>
</tr>
<tr>
<td>duration.inMonths()</td>
<td>duration.inMonths(from :: ANY?, to :: ANY?) :: (DURATION?)</td>
<td>Compute the duration between the ‘from’ instant (inclusive) and the ‘to’ instant (exclusive) in months.</td>
</tr>
<tr>
<td>duration.inSeconds()</td>
<td>duration.inSeconds(from :: ANY?, to :: ANY?) :: (DURATION?)</td>
<td>Compute the duration between the ‘from’ instant (inclusive) and the ‘to’ instant (exclusive) in seconds.</td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>point.distance()</td>
<td>point.distance(from :: POINT?, to :: POINT?) :: (FLOAT?)</td>
<td>Returns a floating point number representing the geodesic distance between any two points in the same CRS.</td>
</tr>
<tr>
<td>point() - Cartesian 2D</td>
<td>point(input :: MAP?) :: (POINT?)</td>
<td>Returns a 2D point object, given two coordinate values in the Cartesian coordinate system.</td>
</tr>
<tr>
<td>point() - Cartesian 3D</td>
<td>point(input :: MAP?) :: (POINT?)</td>
<td>Returns a 3D point object, given three coordinate values in the Cartesian coordinate system.</td>
</tr>
<tr>
<td>point() - WGS 84 2D</td>
<td>point(input :: MAP?) :: (POINT?)</td>
<td>Returns a 2D point object, given two coordinate values in the WGS 84 geographic coordinate system.</td>
</tr>
<tr>
<td>point() - WGS 84 3D</td>
<td>point(input :: MAP?) :: (POINT?)</td>
<td>Returns a 3D point object, given three coordinate values in the WGS 84 geographic coordinate system.</td>
</tr>
<tr>
<td>point.withinBBox()</td>
<td>point.withinBBox(point :: POINT?, lowerLeft :: POINT?, upperRight :: POINT?) :: (BOOLEAN?)</td>
<td>Returns true if the provided point is within the bounding box defined by the two provided points, lowerLeft and upperRight.</td>
</tr>
</tbody>
</table>

### LOAD CSV functions

LOAD CSV functions can be used to get information about the file that is processed by LOAD CSV.

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file()</td>
<td>file() :: (STRING?)</td>
<td>Returns the absolute path of the file that LOAD CSV is using.</td>
</tr>
<tr>
<td>Function</td>
<td>Signature</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>linenumber()</td>
<td>linenumber() :: (INTEGER?)</td>
<td>Returns the line number that LOAD CSV is currently using.</td>
</tr>
</tbody>
</table>

**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. There are two main types of functions that can be developed and used:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Usage</th>
<th>Developing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar</td>
<td>For each row the function takes parameters and returns a result</td>
<td>Using UDF</td>
<td>Extending Neo4j (UDF)</td>
</tr>
<tr>
<td>Aggregating</td>
<td>Consumes many rows and produces an aggregated result</td>
<td>Using aggregating UDF</td>
<td>Extending Neo4j (Aggregating UDF)</td>
</tr>
</tbody>
</table>

**10.1. Predicate functions**

Predicates are boolean functions that return true or false for a given set of non-null input. They are most commonly used to filter out paths in the WHERE part of a query.

Functions:

- all()
- any()
- exists()
- isEmpty()
- none()
- single()
10.1.1. all()

The function `all()` returns `true` if the predicate holds for all elements in the given list. `null` is returned if the list is `null` or all of its elements are `null`.

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

Returns:

A Boolean.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>An expression that returns a list. A single element cannot be explicitly passed as a literal in the cypher statement. However, an implicit conversion will happen for single elements when passing node properties during cypher execution.</td>
</tr>
<tr>
<td>variable</td>
<td>A variable that can be used from within the predicate.</td>
</tr>
<tr>
<td>predicate</td>
<td>A predicate that is tested against all items in the list.</td>
</tr>
</tbody>
</table>

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 a property `age` with a value larger than 30.

Table 255. Result

<table>
<thead>
<tr>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)-[KNOWS,1]-&gt;(2)-[KNOWS,3]-&gt;(3)</td>
</tr>
</tbody>
</table>

Rows: 1

10.1.2. any()

The function `any()` returns `true` if the predicate holds for at least one element in the given list. `null` is returned if the list is `null` or all of its elements are `null`.

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

Returns:

A Boolean.
Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>An expression that returns a list. A single element cannot be explicitly passed as a literal in the cypher statement. However, an implicit conversion will happen for single elements when passing node properties during cypher execution.</td>
</tr>
<tr>
<td>variable</td>
<td>A variable that can be used from within the predicate.</td>
</tr>
<tr>
<td>predicate</td>
<td>A predicate that is tested against all items in the list.</td>
</tr>
</tbody>
</table>

Query

```
MATCH (n)
WHERE any(color IN n.liked_colors WHERE color = 'yellow')
RETURN n
```

The query returns nodes with the property `liked_colors` (as a list), where at least one element has the value 'yellow'.

Table 256. Result

<table>
<thead>
<tr>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node[4]{eyes:&quot;blue&quot;,liked_colors:[&quot;pink&quot;,&quot;yellow&quot;,&quot;black&quot;],name:&quot;Eskil&quot;,age:41}</td>
</tr>
</tbody>
</table>

Rows: 1

10.1.3. `exists()`

The function `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. `null` is returned if the input argument is `null`.

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

**Returns:**

A Boolean.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pattern-or-property</td>
<td>A pattern or a property (in the form 'variable.prop').</td>
</tr>
</tbody>
</table>

Query

```
MATCH (n)
WHERE n.name IS NOT NULL
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 or false) indicating if they are married.

Table 257. Result

<table>
<thead>
<tr>
<th></th>
<th>is_married</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Alice&quot;</td>
<td>false</td>
</tr>
<tr>
<td>&quot;Bob&quot;</td>
<td>true</td>
</tr>
<tr>
<td>&quot;Charlie&quot;</td>
<td>false</td>
</tr>
<tr>
<td>&quot;Daniel&quot;</td>
<td>false</td>
</tr>
<tr>
<td>&quot;Eskil&quot;</td>
<td>false</td>
</tr>
</tbody>
</table>

Rows: 5

Query

MATCH (a), (b)
WHERE exists(a.name) AND NOT exists(b.name)
OPTIONAL MATCH (c:DoesNotExist)
RETURN a.name AS a_name, b.name AS b_name, exists(b.name) AS b_has_name, c.name AS c_name, exists(c.name) AS c_has_name
ORDER BY a_name, b_name, c_name
LIMIT 1

Three nodes are returned: one with a property name, one without a property name, and one that does not exist (e.g., is null). This query exemplifies the behavior of exists() when operating on null nodes.

Table 258. Result

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a_name</td>
<td>b_name</td>
<td>b_has_name</td>
<td>c_name</td>
<td>c_has_name</td>
</tr>
<tr>
<td>&quot;Alice&quot;</td>
<td>&lt;null&gt;</td>
<td>false</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

Rows: 1

Note that the exists() function is deprecated for property input. Please use the IS NOT NULL predicate instead.

10.1.4. isEmpty()

The function isEmpty() returns true if the given list or map contains no elements or if the given string contains no characters.

Syntax: isEmpty(list)

Returns:
A Boolean.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>An expression that returns a list.</td>
</tr>
</tbody>
</table>

Query

```sql
MATCH (n)
WHERE NOT isEmpty(n.liked_colors)
RETURN n
```

The nodes with the property `liked_colors` being non-empty are returned.

Table 259. Result

<table>
<thead>
<tr>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node[5]{eyes: &quot;&quot;, liked_colors: [&quot;blue&quot;, &quot;green&quot;], alias: &quot;Frank&quot;, age: 61}</td>
</tr>
</tbody>
</table>

Rows: 2

Syntax: `isEmpty(map)`

Returns:

A Boolean.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>An expression that returns a map.</td>
</tr>
</tbody>
</table>

Query

```sql
MATCH (n)
WHERE isEmpty(properties(n))
RETURN n
```

Nodes that do not have any properties are returned.

Table 260. Result

<table>
<thead>
<tr>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node[6]{}</td>
</tr>
</tbody>
</table>

Rows: 1

Syntax: `isEmpty(string)`
Returns:

A Boolean.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>An expression that returns a string.</td>
</tr>
</tbody>
</table>

Query

```
MATCH (n)
WHERE isEmpty(n.eyes)
RETURN n.age AS age
```

The age are returned for each node that has a property `eyes` where the value evaluates to be empty (empty string).

**Table 261. Result**

<table>
<thead>
<tr>
<th>age</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>61</td>
<td></td>
</tr>
</tbody>
</table>

Rows: 1

The function `isEmpty()`, like most other Cypher functions, returns `null` if `null` is passed in to the function. That means that a predicate `isEmpty(n.eyes)` will filter out all nodes where the `eyes` property is not set. Thus, `isEmpty()` is not suited to test for null values. `IS NULL` or `IS NOT NULL` should be used for that purpose.

10.1.5. none()

The function `none()` returns `true` if the predicate does not hold for any element in the given list. `null` is returned if the list is `null` or all of its elements are `null`.

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

Returns:

A Boolean.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>An expression that returns a list. A single element cannot be explicitly passed as a literal in the cypher statement. However, an implicit conversion will happen for single elements when passing node properties during cypher execution.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>variable</td>
<td>A variable that can be used from within the predicate.</td>
</tr>
<tr>
<td>predicate</td>
<td>A predicate that is tested against all items in the list.</td>
</tr>
</tbody>
</table>

**Query**

```sql
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 a property `age` with the value 25.

**Table 262. Result**

<table>
<thead>
<tr>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)-[KNOWS,1]-&gt;(2)</td>
</tr>
<tr>
<td>(0)-[KNOWS,1]-&gt;(2)-[KNOWS,3]-&gt;(3)</td>
</tr>
</tbody>
</table>

Rows: 2

**10.1.6. single()**

The function `single()` returns `true` if the predicate holds for exactly one of the elements in the given list. `null` is returned if the list is `null` or all of its elements are `null`.

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

**Returns:**

A Boolean.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>An expression that returns a list.</td>
</tr>
<tr>
<td>variable</td>
<td>A variable that can be used from within the predicate.</td>
</tr>
<tr>
<td>predicate</td>
<td>A predicate that is tested against all items in the list.</td>
</tr>
</tbody>
</table>

**Query**

```sql
MATCH p = (n)-->(b)
WHERE
  n.name = 'Alice'
  AND single(var IN nodes(p) WHERE var.eyes = 'blue')
RETURN p
```

In every returned path there is exactly one node that has a property `eyes` with the value 'blue'.
10.2. Scalar functions

Scalar functions return a single value.

Functions:

- `coalesce()`
- `endNode()`
- `head()`
- `id()`
- `last()`
- `length()`
- `properties()`
- `randomUUID()`
- `size()`
- Size of pattern comprehension
- Size of string
- `startNode()`
- `timestamp()`
- `toBoolean()`
- `toBooleanOrNull()`
- `toFloat()`
- `toFloatOrNull()`
- `toInteger()`
- `toIntegerOrNull()`
- `type()`
The `length()` and `size()` functions are quite similar, and so it is important to take note of the difference.

**Function `length()`**

Only works for paths.

**Function `size()`**

Only works for the three types: strings, lists, and pattern comprehension.

```graph
Developer
  name = 'Alice'
  age = 38
  eyes = 'brown'

  KNOWS

  name = 'Charlie'
  age = 53
  eyes = 'green'

  KNOWS

  name = 'Bob'
  age = 25
  eyes = 'blue'

  KNOWS

  name = 'Daniel'
  age = 54
  eyes = 'brown'

  KNOWS

  eyes = 'blue'
  liked_colors = ['pink', 'yellow', 'black']
  name = 'Eskil'
  age = 41
  MARRIED
```

### 10.2.1. coalesce()

The function `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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>An expression that may return null.</td>
</tr>
</tbody>
</table>

**Considerations:**

null will be returned if all the arguments are null.

**Query**

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

**Table 264. Result**
10.2.2. endNode()

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

**Syntax:** `endNode(relationship)`

**Returns:**
A Node.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>relationship</td>
<td>An expression that returns a relationship.</td>
</tr>
</tbody>
</table>

**Considerations:**

`endNode(null)` returns null.

**Query**

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

**Table 265. Result**

<table>
<thead>
<tr>
<th>endNode(r)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node[2]{name:&quot;Charlie&quot;,age:53,eyes:&quot;green&quot;}</td>
</tr>
<tr>
<td>Node[1]{name:&quot;Bob&quot;,age:25,eyes:&quot;blue&quot;}</td>
</tr>
</tbody>
</table>

Rows: 2

10.2.3. head()

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

**Syntax:** `head(expression)`

**Returns:**

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

**Arguments:**
### expression

An expression that returns a list.

**Considerations:**

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

**Query**

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

The first element in the list is returned.

**Table 266. Result**

<table>
<thead>
<tr>
<th>a.liked_colors</th>
<th>head(a.liked_colors)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[&quot;pink&quot;,&quot;yellow&quot;,&quot;black&quot;]</td>
<td>&quot;pink&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

### 10.2.4. id()

The function `id()` returns a node or a relationship identifier, unique by an object type and a database. Therefore, it is perfectly allowable for `id()` to return the same value for both nodes and relationships in the same database. For examples on how to get a node and a relationship by ID, see Get node or relationship by id.

Neo4j implements the id so that:

**Node**

Every node in a database has an identifier. The identifier for a node is guaranteed to be unique among other nodes' identifiers in the same database, within the scope of a single transaction.

**Relationship**

Every relationship in a database has an identifier. The identifier for a relationship is guaranteed to be unique among other relationships' identifiers in the same database, within the scope of a single transaction.

**Syntax:** `id(expression)`

**Returns:**

An Integer.
Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>An expression that returns a node or a relationship.</td>
</tr>
</tbody>
</table>

Considerations:

- `id(null)` returns `null`.

Query

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

The node identifier for each of the nodes is returned.

Table 267. Result

<table>
<thead>
<tr>
<th>id(a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

Rows: 5

10.2.5. `last()`

The function `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 the list.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>An expression that returns a list.</td>
</tr>
</tbody>
</table>

Considerations:

- `last(null)` returns `null`.

- `last([])` returns `null`.

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

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

The last element in the list is returned.

Table 268. Result

<table>
<thead>
<tr>
<th>a.liked_colors</th>
<th>last(a.liked_colors)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[&quot;pink&quot;,&quot;yellow&quot;,&quot;black&quot;]</td>
<td>&quot;black&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

10.2.6. length()

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

Syntax: `length(path)`

Returns:

An Integer.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>An expression that returns a path.</td>
</tr>
</tbody>
</table>

Considerations:

`length(null)` returns `null`.

Query

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

The length of the path `p` is returned.

Table 269. Result

<table>
<thead>
<tr>
<th>length(p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

Rows: 3
10.2.7. properties()

The function `properties()` returns a map containing all the properties; the function can be utilized for a relationship or a node. If the argument is already a map, it is returned unchanged.

**Syntax:** `properties(expression)`

**Returns:** A Map.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>An expression that returns a relationship, a node, or a map.</td>
</tr>
</tbody>
</table>

**Considerations:**

`properties(null)` returns `null`.

**Query**

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

**Table 270. Result**

```
properties(p)
{city -> "Berlin", name -> "Stefan"}
```

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

10.2.8. randomUUID()

The function `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**

```sql
RETURN randomUUID() AS uuid
```
A randomly-generated UUID is returned.

10.2.9. size()

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

Syntax: `size(list)`

Returns:

An Integer.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>An expression that returns a list.</td>
</tr>
</tbody>
</table>

Considerations:

`size(null)` returns `null`.

Query

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

The number of elements in the list is returned.

10.2.10. size() applied to pattern comprehension

This is the same function `size()` as described above, but you pass in a pattern comprehension. The function size will then calculate on a list of paths.

Syntax: `size(pattern expression)`

Arguments:
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pattern expression</td>
<td>A pattern expression that returns a list.</td>
</tr>
</tbody>
</table>

**Query**

```
MATCH (a)
WHERE a.name = 'Alice'
RETURN size([p=(a)-->(p)-->(p) | p]) AS fos
```

**Table 273. Result**

<table>
<thead>
<tr>
<th>fos</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows:</td>
<td>1</td>
</tr>
</tbody>
</table>

The number of paths matching the pattern expression is returned. (The size of the list of paths).

**10.2.11. size() applied to string**

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

**Syntax:** `size(string)`

**Returns:**

An Integer.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>An expression that returns a string value.</td>
</tr>
</tbody>
</table>

**Considerations:**

`size(null)` returns `null`.

**Query**

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

**Table 274. Result**

<table>
<thead>
<tr>
<th>size(a.name)</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows:</td>
<td>1</td>
</tr>
</tbody>
</table>

The number of characters in the string 'Charlie' is returned.
10.2.12. startNode()

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

**Syntax:** `startNode(relationship)`

**Returns:**

A Node.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>relationship</td>
<td>An expression that returns a relationship.</td>
</tr>
</tbody>
</table>

**Considerations:**

`startNode(null)` returns `null`.

**Query**

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

**Table 275. Result**

<table>
<thead>
<tr>
<th>startNode(r)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node[0]{name:&quot;Alice&quot;,age:38,eyes:&quot;brown&quot;}</td>
</tr>
<tr>
<td>Node[0]{name:&quot;Alice&quot;,age:38,eyes:&quot;brown&quot;}</td>
</tr>
</tbody>
</table>

Rows: 2

10.2.13. timestamp()

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

- **It is the equivalent of `datetime().epochMillis`.**

**Syntax:** `timestamp()`

**Returns:**

An Integer.

**Considerations:**

`timestamp()` will return the same value during one entire query, even for long-running queries.
The time in milliseconds is returned.

Table 276. Result

<table>
<thead>
<tr>
<th>timestamp()</th>
<th>1660050394628</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows:</td>
<td>1</td>
</tr>
</tbody>
</table>

10.2.14. toBoolean()

The function `toBoolean()` converts a string, integer or boolean value to a boolean value.

Syntax: `toBoolean(expression)`

Returns:

A Boolean.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>An expression that returns a boolean, string or integer value.</td>
</tr>
</tbody>
</table>

Considerations:

- `toBoolean(null)` returns `null`.
- If `expression` is a boolean value, it will be returned unchanged.
- If the parsing fails, `null` will be returned.
- If `expression` is the integer value 0, `false` will be returned. For any other integer value `true` will be returned.
- This function will return an error if provided with an expression that is not a string, integer or boolean value.

Query

```sql
RETURN toBoolean('true'), toBoolean('not a boolean'), toBoolean(0)
```

Table 277. Result

<table>
<thead>
<tr>
<th>toBoolean('true')</th>
<th>toBoolean('not a boolean')</th>
<th>toBoolean(0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>&lt;null&gt;</td>
<td>false</td>
</tr>
<tr>
<td>Rows:</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
10.2.15. `toBooleanOrNull()`

The function `toBooleanOrNull()` converts a string, integer or boolean value to a boolean value. For any other input value, `null` will be returned.

**Syntax:** `toBooleanOrNull(expression)`

**Returns:**
A Boolean or `null`.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>Any expression that returns a value.</td>
</tr>
</tbody>
</table>

**Considerations:**

- `toBooleanOrNull(null)` returns `null`.
- If `expression` is a boolean value, it will be returned unchanged.
- If the parsing fails, `null` will be returned.
- If `expression` is the integer value `0`, `false` will be returned. For any other integer value `true` will be returned.
- If the `expression` is not a string, integer or boolean value, `null` will be returned.

**Query**

```
RETURN toBooleanOrNull('true'), toBooleanOrNull('not a boolean'), toBooleanOrNull(0), toBooleanOrNull(1.5)
```

**Table 278. Result**

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>true</code></td>
<td><code>true</code></td>
</tr>
<tr>
<td><code>not a boolean</code></td>
<td><code>&lt;null&gt;</code></td>
</tr>
<tr>
<td><code>0</code></td>
<td><code>false</code></td>
</tr>
<tr>
<td><code>1.5</code></td>
<td><code>&lt;null&gt;</code></td>
</tr>
</tbody>
</table>

Rows: 1

10.2.16. `toFloat()`

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

**Syntax:** `toFloat(expression)`

**Returns:**
A Float.

**Arguments:**
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>An expression that returns a numeric or a string value.</td>
</tr>
</tbody>
</table>

**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.
- This function will return an error if provided with an expression that is not an integer, floating point or a string value.

**Query**

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

**Table 279. Result**

<table>
<thead>
<tr>
<th>toFloat('11.5')</th>
<th>toFloat('not a number')</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.5</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

Rows: 1

**10.2.17. toFloatOrNull()**

The function `toFloatOrNull()` converts an integer, floating point or a string value to a floating point number. For any other input value, `null` will be returned.

**Syntax:** `toFloatOrNull(expression)`

**Returns:**

A Float or `null`.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>Any expression that returns a value.</td>
</tr>
</tbody>
</table>

**Considerations:**

- `toFloatOrNull(null)` returns `null`.
- If `expression` is a floating point number, it will be returned unchanged.
- If the parsing fails, `null` will be returned.
- If the `expression` is not an integer, floating point or a string value, `null` will be returned.
Query

```
RETURN toFloatOrNull('11.5'), toFloatOrNull('not a number'), toFloatOrNull(true)
```

Table 280. Result

<table>
<thead>
<tr>
<th>toFloatOrNull('11.5')</th>
<th>toFloatOrNull('not a number')</th>
<th>toFloatOrNull(true)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.5</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

Rows: 1

10.2.18. toInteger()

The function `toInteger()` converts a boolean, integer, floating point or a string value to an integer value.

Syntax: `toInteger(expression)`

Returns: An Integer.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>An expression that returns a boolean, numeric or a string value.</td>
</tr>
</tbody>
</table>

Considerations:

- `toInteger(null)` returns `null`.
- If `expression` is an integer value, it will be returned unchanged.
- If the parsing fails, `null` will be returned.
- If `expression` is the boolean value `false`, 0 will be returned. If `expression` is the boolean value `true`, 1 will be returned.
- This function will return an error if provided with an expression that is not a boolean, floating point, integer or a string value.

Query

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

Table 281. Result

<table>
<thead>
<tr>
<th>toInteger('42')</th>
<th>toInteger('not a number')</th>
<th>toInteger(true)</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>&lt;null&gt;</td>
<td>1</td>
</tr>
</tbody>
</table>

Rows: 1
10.2.19. toIntegerOrNull()

The function `toIntegerOrNull()` converts a boolean, integer, floating point or a string value to an integer value. For any other input value, `null` will be returned.

**Syntax:** `toIntegerOrNull(expression)`

**Returns:**

An Integer or `null`.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>Any expression that returns a value.</td>
</tr>
</tbody>
</table>

**Considerations:**

- `toIntegerOrNull(null)` returns `null`.
- If `expression` is an integer value, it will be returned unchanged.
- If the parsing fails, `null` will be returned.
- If `expression` is the boolean value `false`, `0` will be returned. If `expression` is the boolean value `true`, `1` will be returned.
- If the `expression` is not a boolean, floating point, integer or a string value, `null` will be returned.

**Query**

```sql
RETURN toIntegerOrNull('42'), toIntegerOrNull('not a number'), toIntegerOrNull(true), toIntegerOrNull(['A', 'B', 'C'])
```

**Table 282. Result**

<table>
<thead>
<tr>
<th>toIntegerOrNull('42')</th>
<th>toIntegerOrNull('not a number')</th>
<th>toIntegerOrNull(true)</th>
<th>toIntegerOrNull(['A', 'B', 'C'])</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>&lt;null&gt;</td>
<td>1</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

Rows: 1

10.2.20. type()

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

**Syntax:** `type(relationship)`

**Returns:**

A String.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>relationship</td>
<td>An expression that returns a relationship.</td>
</tr>
</tbody>
</table>

**Considerations:**

```
type(null) returns null.
```

**Query**

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

The relationship type of \( r \) is returned.

**Table 283. Result**

<table>
<thead>
<tr>
<th>type(r)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;KNOWS&quot;</td>
</tr>
<tr>
<td>&quot;KNOWS&quot;</td>
</tr>
</tbody>
</table>

Rows: 2

### 10.3. Aggregating functions

*Aggregating functions take a set of values and calculate an aggregated value over them.*

**Functions:**

- \texttt{avg()} - Numeric values
- \texttt{avg()} - Durations
- \texttt{collect()}
- \texttt{count()}
- \texttt{max()}
- \texttt{min()}
- \texttt{percentileCont()}
- \texttt{percentileDisc()}
- \texttt{stDev()}
- \texttt{stDevP()}
- \texttt{sum()} - Numeric values
- \texttt{sum()} - Durations

Aggregation can be computed over all the matching paths, or it can be further divided by introducing grouping keys. Grouping keys 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, so it will be the grouping key. The latter, `count(*)` is an aggregate expression. The matching paths 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 in `Syntax → Aggregation operators`.

The following graph is used for the examples below:

![Graph](image-url)

Graph

10.3.1. `avg()` - Numeric values

The function `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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>expression</code></td>
<td>An expression returning a set of numeric values.</td>
</tr>
</tbody>
</table>

**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 284. Result

<table>
<thead>
<tr>
<th>avg(n.age)</th>
<th>30.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

10.3.2. avg() - Durations

The function avg() returns the average of a set of Durations.

Syntax: avg(expression)

Returns:

A Duration.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>An expression returning a set of Durations.</td>
</tr>
</tbody>
</table>

Considerations:

Any null values are excluded from the calculation.

avg(null) returns null.

Query

```
UNWIND [duration('P2DT3H'), duration('PT1H45S')] AS dur
RETURN avg(dur)
```

The average of the two supplied Durations is returned.

Table 285. Result

<table>
<thead>
<tr>
<th>avg(dur)</th>
<th>P1DT2H22.5S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>
10.3.3. collect()

The function `collect()` returns a single aggregated list containing the values returned by an expression.

**Syntax:** `collect(expression)`

**Returns:**

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

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>An expression returning a set of values.</td>
</tr>
</tbody>
</table>

**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 286. Result**

<table>
<thead>
<tr>
<th><code>collect(n.age)</code></th>
<th><code>[13, 33, 44]</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows:</td>
<td>1</td>
</tr>
</tbody>
</table>

10.3.4. count()

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

- `count(*)`  
  returns the number of matching rows.

- `count(expr)`  
  returns the number of non-null values returned by an expression.

**Syntax:** `count(expression)`

**Returns:**

An Integer.
Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>An expression.</td>
</tr>
</tbody>
</table>

Considerations:

- `count(*)` includes rows returning `null`.
- `count(expr)` ignores `null` values.
- `count(null)` returns 0.

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

The function `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 287. Result

<table>
<thead>
<tr>
<th>labels(n)</th>
<th>n.age</th>
<th>count(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Person&quot;</td>
<td>13</td>
<td>4</td>
</tr>
</tbody>
</table>

Rows: 1

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

The function `count(*)` can be used to group the type of matched relationships and return the number.

Query

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

The type of matched relationships are grouped and the group count are returned.

Table 288. Result

<table>
<thead>
<tr>
<th>type(r)</th>
<th>count(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;KNOWS&quot;</td>
<td>3</td>
</tr>
<tr>
<td>&quot;READS&quot;</td>
<td>1</td>
</tr>
</tbody>
</table>

Rows: 2
Counting non-null values

Instead of simply returning the number of rows with count(*), the function 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 nodes with the label Person and a property age is returned. (If you want the sum, use sum(n.age))

Table 289. Result

<table>
<thead>
<tr>
<th>count(n.age)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Rows: 1

Counting with and without duplicates

In this example we are trying to find all our friends of friends, and count them:

```
count(DISTINCT friend_of_friend)
```

Will only count a friend_of_friend once, as DISTINCT removes the duplicates.

```
count(friend_of_friend)
```

Will consider the same friend_of_friend multiple times.

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

Table 290. Result

<table>
<thead>
<tr>
<th>count(DISTINCT friend_of_friend)</th>
<th>count(friend_of_friend)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Rows: 1

10.3.5. max()

The function 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>An expression returning a set containing any combination of property types and lists thereof.</td>
</tr>
</tbody>
</table>

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.

The value '99' (a string), is considered to be a lower value than 1 (an integer), because '99' is a string.

Table 291. Result

<table>
<thead>
<tr>
<th>max(val)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Rows: 1

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 292. Result

<table>
<thead>
<tr>
<th>max(val)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,2]</td>
</tr>
</tbody>
</table>

Rows: 1
Query

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

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

<table>
<thead>
<tr>
<th>Table 293. Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>max(n.age)</td>
</tr>
<tr>
<td>44</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

### 10.3.6. min()

The function `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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>An expression returning a set containing any combination of <em>property types</em> and lists thereof.</td>
</tr>
</tbody>
</table>

**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 294. Result |
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 295. Result**

<table>
<thead>
<tr>
<th>min(val)</th>
</tr>
</thead>
<tbody>
<tr>
<td>['a', 'c', 23]</td>
</tr>
</tbody>
</table>

Tables 1 and 2 provide examples of the MIN function. The MIN function returns the smallest value in a list, set, or collection of values. It is commonly used to find the minimum value in a collection of data, such as the minimum age in a list of people. Tables 1 and 2 provide examples of how to use the MIN function in a query.

**10.3.7. percentileCont()**

The function 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:**
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression.</td>
</tr>
<tr>
<td>percentile</td>
<td>A numeric value between 0.0 and 1.0</td>
</tr>
</tbody>
</table>

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.

Table 297. Result

<table>
<thead>
<tr>
<th>percentileCont(n.age, 0.4)</th>
<th>29.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

10.3.8. percentileDisc()

The function 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression.</td>
</tr>
<tr>
<td>percentile</td>
<td>A numeric value between 0.0 and 1.0</td>
</tr>
</tbody>
</table>

Considerations:

Any null values are excluded from the calculation.

percentileDisc(null, percentile) returns null.
The 50th percentile of the values in the property age is returned.

Table 298. Result

<table>
<thead>
<tr>
<th>percentileDisc(n.age, 0.5)</th>
<th>33</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

10.3.9. stDev()

The function 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression.</td>
</tr>
</tbody>
</table>

Considerations:

Any null values are excluded from the calculation.

stDev(null) returns 0.

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

Table 299. Result

<table>
<thead>
<tr>
<th>stDev(n.age)</th>
<th>15.716233645501712</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>
10.3.10. stDevP()

The function `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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression.</td>
</tr>
</tbody>
</table>

**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 300. Result**

<table>
<thead>
<tr>
<th>stDevP(n.age)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.832251036613439</td>
</tr>
</tbody>
</table>

Rows: 1

10.3.11. sum() - Numeric values

The function `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:**
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>An expression returning a set of numeric values.</td>
</tr>
</tbody>
</table>

**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 301. Result**

<table>
<thead>
<tr>
<th>sum(n.age)</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows:</td>
<td>1</td>
</tr>
</tbody>
</table>

10.3.12. sum() - Durations

The function sum() returns the sum of a set of durations.

**Syntax:** `sum(expression)`

**Returns:**

A Duration.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>An expression returning a set of Durations.</td>
</tr>
</tbody>
</table>

**Considerations:**

Any null values are excluded from the calculation.

**Query**

```
UNWIND [duration('P2DT3H'), duration('PT1H45S')] AS dur
RETURN sum(dur)
```

The sum of the two supplied Durations is returned.

**Table 302. Result**
10.4. List functions

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

Further details and examples of lists may be found in Lists and List operators.

Functions:

- keys()
- labels()
- nodes()
- range()
- reduce()
- relationships()
- reverse()
- tail()
- toBooleanList()
- toFloatList()
- toIntegerList()
- toStringList()

Graph

10.4.1. 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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>An expression that returns a node, a relationship, or a map.</td>
</tr>
</tbody>
</table>

**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 303. Result**

<table>
<thead>
<tr>
<th>keys(a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[&quot;name&quot;,&quot;age&quot;,&quot;eyes&quot;]</td>
</tr>
</tbody>
</table>

Rows: 1

### 10.4.2. 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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>An expression that returns a single node.</td>
</tr>
</tbody>
</table>

**Considerations:**

`labels(null)` returns `null`. 255
Query

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

A list containing all the labels of the node bound to \texttt{a} is returned.

Table 304. Result

<table>
<thead>
<tr>
<th>labels(a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[&quot;Person&quot;,&quot;Developer&quot;]</td>
</tr>
</tbody>
</table>

Rows: 1

10.4.3. \texttt{nodes()}\n
\texttt{nodes()} returns a list containing all the nodes in a path.

Syntax: \texttt{nodes(path)}

Returns:

A list containing Node elements.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>An expression that returns a path.</td>
</tr>
</tbody>
</table>

Considerations:

\texttt{nodes(null)} returns \texttt{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 \texttt{p} is returned.

Table 305. Result

<table>
<thead>
<tr>
<th>nodes(p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Node[0]{name:&quot;Alice&quot;,age:38,eyes:&quot;brown&quot;},Node[1]{name:&quot;Bob&quot;,age:25,eyes:&quot;blue&quot;},Node[4]{eyes:&quot;blue&quot;,array:[:&quot;one&quot;,&quot;two&quot;,&quot;three&quot;],name:&quot;Eskil&quot;,age:41}]</td>
</tr>
</tbody>
</table>

Rows: 1

256
10.4.4. 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. To create ranges with decreasing integer values, use a negative value `step`. The range is inclusive for non-empty ranges, and the arithmetic progression will therefore always contain `start` and — depending on the values of `start`, `step` and `end` — `end`. The only exception where the range does not contain `start` are empty ranges. An empty range will be returned if the value `step` is negative and `start - end` is positive, or vice versa, e.g. `range(0, 5, -1)`.

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

**Returns:**

A list of Integer elements.

**Arguments:**

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

**Query**

```
RETURN range(0, 10), range(2, 18, 3), range(0, 5, -1)
```

Three lists of numbers in the given ranges are returned.

**Table 306. Result**

<table>
<thead>
<tr>
<th>range(0, 10)</th>
<th>range(2, 18, 3)</th>
<th>range(0, 5, -1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0,1,2,3,4,5,6,7,8,9,10]</td>
<td>[2,5,8,11,14,17]</td>
<td>[]</td>
</tr>
</tbody>
</table>

Rows: 1

10.4.5. 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 expression.

Arguments:

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

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 307. Result

<table>
<thead>
<tr>
<th>reduction</th>
<th>117</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

10.4.6. relationships()

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

Syntax: relationships(path)

Returns:

A list containing Relationship elements.

 Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>An expression that returns a path.</td>
</tr>
</tbody>
</table>

Considerations:

relationships(null) returns null.
A list containing all the relationships in the path $p$ is returned.

### Table 308. Result

<table>
<thead>
<tr>
<th>relationships(p)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[:KNOWS[0]{},:MARRIED[4]{}]</td>
<td></td>
</tr>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

#### 10.4.7. 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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>original</code></td>
<td>An expression that returns a list.</td>
</tr>
</tbody>
</table>

**Considerations:**

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

### Query

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

### Table 309. Result

<table>
<thead>
<tr>
<th>reverse(ids)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[487,&lt;null&gt;,521,&quot;abc&quot;,4923]</td>
<td></td>
</tr>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

#### 10.4.8. tail()

`tail()` returns a list $l_{\text{result}}$ containing all the elements, excluding the first one, from a list $l$. 

---

259
Syntax: \texttt{tail(list)}

Returns:

A list containing heterogeneous elements; the types of the elements are determined by the elements in \texttt{list}.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>An expression that returns a list.</td>
</tr>
</tbody>
</table>

Query

\texttt{MATCH (a) WHERE a.name = 'Eskil'\nRETURN a.array, tail(a.array)}

The property named \texttt{array} and a list comprising all but the first element of the \texttt{array} property are returned.

Table 310. Result

<table>
<thead>
<tr>
<th>a.array</th>
<th>tail(a.array)</th>
</tr>
</thead>
<tbody>
<tr>
<td>['one','two','three']</td>
<td>['two','three']</td>
</tr>
</tbody>
</table>

Rows: 1

10.4.9. \texttt{toBooleanList()}

\texttt{toBooleanList()} converts a list of values and returns a list of boolean values. If any values are not convertible to boolean they will be null in the list returned.

Syntax: \texttt{toBooleanList(list)}

Returns:

A list containing the converted elements; depending on the input value a converted value is either a boolean value or \texttt{null}.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>An expression that returns a list.</td>
</tr>
</tbody>
</table>

Considerations:

- Any \texttt{null} element in \texttt{list} is preserved.
- Any boolean value in \texttt{list} is preserved.
- If the \texttt{list} is \texttt{null}, \texttt{null} will be returned.
- If the \texttt{list} is not a list, an error will be returned.
The conversion for each value in list is done according to the toBooleanOrNull() function.

Query

```
RETURN toBooleanList(null) as noList,
    toBooleanList([null, null]) as nullsInList,
    toBooleanList(["a string", true, "false", null, ["A", "B"]]) as mixedList
```

Table 311. Result

<table>
<thead>
<tr>
<th></th>
<th>nullsInList</th>
<th>mixedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;null&gt;</td>
<td>[&lt;null&gt;,&lt;null&gt;]</td>
<td>[&lt;null&gt;,true,false,&lt;null&gt;,&lt;null&gt;]</td>
</tr>
</tbody>
</table>

Rows: 1

10.4.10. toFloatList()

toFloatList() converts a list of values and returns a list of floating point values. If any values are not convertible to floating point they will be null in the list returned.

Syntax: toFloatList(list)

Returns:

A list containing the converted elements; depending on the input value a converted value is either a floating point value or null.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>An expression that returns a list.</td>
</tr>
</tbody>
</table>

Considerations:

Any null element in list is preserved.

Any floating point value in list is preserved.

If the list is null, null will be returned.

If the list is not a list, an error will be returned.

The conversion for each value in list is done according to the toFloatOrNull() function.

Query

```
RETURN toFloatList(null) as noList,
    toFloatList([null, null]) as nullsInList,
    toFloatList(["a string", 2.5, "3.14159", null, ["A", "B"]]) as mixedList
```

Table 312. Result
10.4.11. toIntegerList()

toIntegerList() converts a list of values and returns a list of integer values. If any values are not convertible to integer they will be null in the list returned.

Syntax: toIntegerList(list)

Returns:

A list containing the converted elements; depending on the input value a converted value is either a integer value or null.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>An expression that returns a list.</td>
</tr>
</tbody>
</table>

Considerations:

- Any null element in list is preserved.
- Any integer value in list is preserved.
- If the list is null, null will be returned.
- If the list is not a list, an error will be returned.
- The conversion for each value in list is done according to the toIntegerOrNull() function.

Query

```
RETURN toIntegerList(null) as noList,
toIntegerList([null, null]) as nullsInList,
toIntegerList(['a string', 2, '5', null, ['A','B']]) as mixedList
```

Table 313. Result

<table>
<thead>
<tr>
<th>noList</th>
<th>nullsInList</th>
<th>mixedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;null&gt;</td>
<td>[&lt;null&gt;,&lt;null&gt;]</td>
<td>[&lt;null&gt;,2,5.3.14159,&lt;null&gt;,&lt;null&gt;]</td>
</tr>
</tbody>
</table>

Rows: 1

10.4.12. toStringList()

toStringList() converts a list of values and returns a list of string values. If any values are not convertible to string they will be null in the list returned.

Syntax: toStringList(list)
Returns:

A list containing the converted elements; depending on the input value a converted value is either a string value or \texttt{null}.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{list}</td>
<td>An expression that returns a list.</td>
</tr>
</tbody>
</table>

Considerations:

- Any \texttt{null} element in \texttt{list} is preserved.
- Any string value in \texttt{list} is preserved.
- If the \texttt{list} is \texttt{null}, \texttt{null} will be returned.
- If the \texttt{list} is not a list, an error will be returned.
- The conversion for each value in \texttt{list} is done according to the \texttt{toStringOrNull()} function.

Query

\begin{verbatim}
RETURN toStringList(null) as noList, toStringList([null, null]) as nullsInList, toStringList(['already a string', 2, date({year:1955, month:11, day:5}), null, ['A', 'B']]) as mixedList
\end{verbatim}

Table 314. Result

<table>
<thead>
<tr>
<th>noList</th>
<th>nullsInList</th>
<th>mixedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;null&gt;</td>
<td>[&lt;null&gt;,&lt;null&gt;]</td>
<td>['already a string','2','1955-11-05',&lt;null&gt;,&lt;null&gt;]</td>
</tr>
</tbody>
</table>

Rows: 1

10.5. 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:

- \texttt{abs()}
- \texttt{ceil()}
- \texttt{floor()}
- \texttt{rand()}
- \texttt{round()}
- \texttt{round()}, with precision
- `round()`, with precision and rounding mode
- `sign()`

The following graph is used for the examples below:

![Graph](image)

**Graph**

10.5.1. `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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression.</td>
</tr>
</tbody>
</table>

**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 315. Result**
10.5.2. 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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>expression</code></td>
<td>A numeric expression.</td>
</tr>
</tbody>
</table>

**Considerations:**

`ceil(null)` returns `null`.

**Query**

```plaintext
RETURN ceil(0.1)
```

The ceil of `0.1` is returned.

**Table 316. Result**

<table>
<thead>
<tr>
<th><code>ceil(0.1)</code></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>1.0</code></td>
<td></td>
</tr>
</tbody>
</table>

Rows: 1

10.5.3. 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression.</td>
</tr>
</tbody>
</table>

Considerations:

floor(null) returns null.

Query

```
RETURN floor(0.9)
```

The floor of 0.9 is returned.

Table 317. Result

<table>
<thead>
<tr>
<th>floor(0.9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
</tr>
</tbody>
</table>

10.5.4. 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 318. Result

<table>
<thead>
<tr>
<th>rand()</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9805631841924222</td>
</tr>
</tbody>
</table>

10.5.5. round()

round() returns the value of the given number rounded to the nearest integer, with half-way values always rounded up.
**Syntax:** `round(expression)`

**Returns:**

A Float.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression to be rounded.</td>
</tr>
</tbody>
</table>

**Considerations:**

`round(null)` returns `null`.

**Query**

```sql
RETURN round(3.141592)
```

3.0 is returned.

**Table 319. Result**

<table>
<thead>
<tr>
<th>round(3.141592)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.0</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

**10.5.6. `round()`, with precision**

`round()` returns the value of the given number rounded with the specified precision, with half-values always being rounded up.

**Syntax:** `round(expression, precision)`

**Returns:**

A Float.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression to be rounded.</td>
</tr>
<tr>
<td>precision</td>
<td>A numeric expression specifying precision.</td>
</tr>
</tbody>
</table>

**Considerations:**

`round(null)` returns `null`. 
Query

```
RETURN round(3.141592, 3)
```

3.142 is returned.

Table 320. Result

<table>
<thead>
<tr>
<th>expression</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>round(3.141592, 3)</td>
<td>3.142</td>
</tr>
</tbody>
</table>

10.5.7. round(), with precision and rounding mode

`round()` returns the value of the given number rounded with the specified precision and the specified rounding mode.

**Syntax:** `round(expression, precision, mode)`

**Returns:** A Float.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression to be rounded.</td>
</tr>
<tr>
<td>precision</td>
<td>A numeric expression specifying precision.</td>
</tr>
<tr>
<td>mode</td>
<td>A string expression specifying rounding mode.</td>
</tr>
</tbody>
</table>

**Modes:**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEILING</td>
<td>Round towards positive infinity.</td>
</tr>
<tr>
<td>DOWN</td>
<td>Round towards zero.</td>
</tr>
<tr>
<td>FLOOR</td>
<td>Round towards zero.</td>
</tr>
<tr>
<td>HALF_DOWN</td>
<td>Round towards closest value of given precision, with half-values always being rounded down.</td>
</tr>
<tr>
<td>HALF_EVEN</td>
<td>Round towards closest value of given precision, with half-values always being rounded to the even neighbor.</td>
</tr>
<tr>
<td>HALF_UP</td>
<td>Round towards closest value of given precision, with half-values always being rounded up.</td>
</tr>
<tr>
<td>UP</td>
<td>Round away from zero.</td>
</tr>
</tbody>
</table>
Considerations:

round(null) returns null.

Query

RETURN round(3.141592, 2, 'CEILING')

3.15 is returned.

Table 321. Result

<table>
<thead>
<tr>
<th>expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.141592, 2, 'CEILING'</td>
<td>3.15</td>
</tr>
</tbody>
</table>

10.5.8. 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression.</td>
</tr>
</tbody>
</table>

Considerations:

sign(null) returns null.

Query

RETURN sign(-17), sign(0.1)

The signs of -17 and 0.1 are returned.

Table 322. Result

<table>
<thead>
<tr>
<th>sign(-17)</th>
<th>sign(0.1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>1</td>
</tr>
</tbody>
</table>

Rows: 1
10.6. 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()

10.6.1. e()

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

Syntax: e()

Returns: A Float.

Query

```
RETURN e()
```

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

Table 323. Result

<table>
<thead>
<tr>
<th>e()</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.718281828459045</td>
</tr>
</tbody>
</table>

Rows: 1

10.6.2. 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression.</td>
</tr>
</tbody>
</table>

Considerations:

\[ \text{exp(null) returns null.} \]

Query

```
RETURN exp(2)
```

\[ e \text{ to the power of 2 is returned.} \]

Table 324. Result

<table>
<thead>
<tr>
<th>exp(2)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7.38905609893065</td>
<td></td>
</tr>
</tbody>
</table>

10.6.3. \text{log()}

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

Syntax: \( \log(\text{expression}) \)

Returns:

A Float.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression.</td>
</tr>
</tbody>
</table>

Considerations:

\[ \log(null) \text{ returns null.} \]

\[ \log(0) \text{ returns null.} \]

Query

```
RETURN log(27)
```

The natural logarithm of 27 is returned.

Table 325. Result
10.6.4. log10()

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

**Syntax:** log10(expression)

**Returns:**

A Float.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression.</td>
</tr>
</tbody>
</table>

**Considerations:**

- log10(null) returns null.
- log10(0) returns null.

**Query**

```
RETURN log10(27)
```

The common logarithm of 27 is returned.

**Table 326. Result**

<table>
<thead>
<tr>
<th>log10(27)</th>
<th>1.4313637641589874</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

10.6.5. sqrt()

sqrt() returns the square root of a number.

**Syntax:** sqrt(expression)

**Returns:**

A Float.

**Arguments:**

272
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression.</td>
</tr>
</tbody>
</table>

**Considerations:**

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

**Query**

```
RETURN sqrt(256)
```

The square root of 256 is returned.

**Table 327. Result**

<table>
<thead>
<tr>
<th>sqrt(256)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.0</td>
</tr>
</tbody>
</table>

Rows: 1

### 10.7. 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()`
10.7.1. acos()

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

Syntax: acos(expression)

Returns:

A Float.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression that represents the angle in radians.</td>
</tr>
</tbody>
</table>

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 328. Result

<table>
<thead>
<tr>
<th>acos(0.5)</th>
<th>1.0471975511965979</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

10.7.2. asin()

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

Syntax: asin(expression)

Returns:

A Float.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression that represents the angle in radians.</td>
</tr>
</tbody>
</table>
Considerations:

**asin(null)** returns null.

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

<table>
<thead>
<tr>
<th>Query</th>
</tr>
</thead>
</table>
| ```
RETURN asin(0.5)
``` |

The arcsine of 0.5 is returned.

**Table 329. Result**

<table>
<thead>
<tr>
<th>asin(0.5)</th>
<th>0.5235987755982989</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

### 10.7.3. atan()

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

**Syntax:** atan(expression)

**Returns:**

A Float.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression that represents the angle in radians.</td>
</tr>
</tbody>
</table>

**Considerations:**

atan(null) returns null.

<table>
<thead>
<tr>
<th>Query</th>
</tr>
</thead>
</table>
| ```
RETURN atan(0.5)
``` |

The arctangent of 0.5 is returned.

**Table 330. Result**

<table>
<thead>
<tr>
<th>atan(0.5)</th>
<th>0.4636476090008061</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>
10.7.4. atan2()

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

Syntax: atan2(expression1, expression2)

Returns:

A Float.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression1</td>
<td>A numeric expression for y that represents the angle in radians.</td>
</tr>
<tr>
<td>expression2</td>
<td>A numeric expression for x that represents the angle in radians.</td>
</tr>
</tbody>
</table>

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 331. Result

<table>
<thead>
<tr>
<th>atan2(0.5, 0.6)</th>
<th>0.6947382761967033</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

10.7.5. cos()

cos() returns the cosine of a number.

Syntax: cos(expression)

Returns:

A Float.

Arguments:
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression that represents the angle in radians.</td>
</tr>
</tbody>
</table>

Considerations:

\[
\cos(\text{null}) \text{ returns } \text{null}. \]

Query

\[
\text{RETURN } \cos(0.5) \]

The cosine of 0.5 is returned.

Table 332. Result

<table>
<thead>
<tr>
<th>cos(0.5)</th>
<th>0.8775825618903728</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows:</td>
<td>1</td>
</tr>
</tbody>
</table>

10.7.6. \text{cot(\textit{}\textit{)}}

\text{cot(\textit{}\textit{)}} \text{ returns the cotangent of a number.}

Syntax: \text{cot(expression)}

Returns:

A Float.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression that represents the angle in radians.</td>
</tr>
</tbody>
</table>

Considerations:

\[
\cot(\text{null}) \text{ returns } \text{null}. \]

\[
\cot(0) \text{ returns } \text{null}. \]

Query

\[
\text{RETURN } \cot(0.5) \]

The cotangent of 0.5 is returned.

Table 333. Result
10.7.7. degrees()

_degrees() converts radians to degrees.

**Syntax:** _degrees(expression)_

**Returns:**

A Float.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression that represents the angle in radians.</td>
</tr>
</tbody>
</table>

**Considerations:**

_degrees(null) returns null.

**Query**

```sql
RETURN degrees(3.14159)
```

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

**Table 334. Result**

<table>
<thead>
<tr>
<th>degrees(3.14159)</th>
<th>179.9998479605043</th>
</tr>
</thead>
</table>

Rows: 1

10.7.8. haversin()

_haversin()_ returns half the versine of a number.

**Syntax:** _haversin(expression)_

**Returns:**

A Float.

**Arguments:**
### Considerations:

**haversin(null)** returns **null**.

**Query**

```
RETURN haversin(0.5)
```

The haversine of **0.5** is returned.

**Table 335. Result**

<table>
<thead>
<tr>
<th>haversin(0.5)</th>
<th>0.06120871905481362</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows:</td>
<td>1</td>
</tr>
</tbody>
</table>

### 10.7.9. 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 336. Result**

<table>
<thead>
<tr>
<th>dist</th>
<th>9129.969740051658</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows:</td>
<td>1</td>
</tr>
<tr>
<td>Nodes created:</td>
<td>2</td>
</tr>
<tr>
<td>Properties set:</td>
<td>4</td>
</tr>
<tr>
<td>Labels added:</td>
<td>2</td>
</tr>
</tbody>
</table>

### 10.7.10. **pi()**

**pi()** returns the mathematical constant **pi**.

**Syntax:** **pi()**
Returns:

A Float.

Query

```
RETURN pi()
```

The constant pi is returned.

Table 337. Result

<table>
<thead>
<tr>
<th>expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pi()</td>
<td>3.141592653589793</td>
</tr>
</tbody>
</table>

10.7.11. radians()

radians() converts degrees to radians.

Syntax: radians(expression)

Returns:

A Float.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression that represents the angle in degrees.</td>
</tr>
</tbody>
</table>

Considerations:

radians(null) returns null.

Query

```
RETURN radians(180)
```

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

Table 338. Result

<table>
<thead>
<tr>
<th>radians(180)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.141592653589793</td>
<td></td>
</tr>
</tbody>
</table>
10.7.12. sin()

\texttt{sin()} returns the sine of a number.

**Syntax:** \texttt{sin(expression)}

**Returns:**

A Float.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression that represents the angle in radians.</td>
</tr>
</tbody>
</table>

**Considerations:**

\texttt{sin(null)} returns \texttt{null}. 

**Query**

```
RETURN sin(0.5)
```

The sine of 0.5 is returned.

**Table 339. Result**

<table>
<thead>
<tr>
<th>sin(0.5)</th>
<th>0.479425538604203</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

10.7.13. tan()

tan() returns the tangent of a number.

**Syntax:** \texttt{tan(expression)}

**Returns:**

A Float.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression that represents the angle in radians.</td>
</tr>
</tbody>
</table>

**Considerations:**
tan(null) returns null.

Query

```
RETURN tan(0.5)
```

The tangent of 0.5 is returned.

### Table 340. Result

<table>
<thead>
<tr>
<th>tan(0.5)</th>
<th>0.5463024898437905</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

#### 10.8. 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, the `size()` function applied to any Unicode character will return 1, even if the character does not fit in the 16 bits of one `char`.

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 format.

See also **String operators**.

Functions:

- `left()`
- `lTrim()`
- `replace()`
- `reverse()`
- `right()`
- `rTrim()`
- `split()`
- `substring()`
- `toLower()`
- `toString()`
10.8.1. left()

left() returns a string containing the specified number of leftmost characters of the original string.

Syntax: `left(original, length)`

Returns:

A String.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>original</code></td>
<td>An expression that returns a string.</td>
</tr>
<tr>
<td><code>n</code></td>
<td>An expression that returns a positive integer.</td>
</tr>
</tbody>
</table>

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 341. Result

<table>
<thead>
<tr>
<th><code>left('hello', 3)</code></th>
<th>&quot;hel&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

10.8.2. ltrim()

ltrim() returns the original string with leading whitespace removed.

Syntax: `ltrim(original)`

Returns:
A String.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>An expression that returns a string.</td>
</tr>
</tbody>
</table>

Considerations:

lTrim(null) returns null

Query

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

Table 342. Result

| lTrim(' hello') | "hello" | Rows: 1 |

10.8.3. 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>An expression that returns a string.</td>
</tr>
<tr>
<td>search</td>
<td>An expression that specifies the string to be replaced in original.</td>
</tr>
<tr>
<td>replace</td>
<td>An expression that specifies the replacement string.</td>
</tr>
</tbody>
</table>

Considerations:

If any argument is null, null will be returned.

If search is not found in original, original will be returned.
**Query**

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

**Table 343. Result**

<table>
<thead>
<tr>
<th>replace(&quot;hello&quot;, &quot;l&quot;, &quot;w&quot;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;hewwo&quot;</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

### 10.8.4. 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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>An expression that returns a string.</td>
</tr>
</tbody>
</table>

**Considerations:**

`reverse(null)` returns null.

**Query**

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

**Table 344. Result**

<table>
<thead>
<tr>
<th>reverse('anagram')</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;margana&quot;</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

### 10.8.5. right()

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

**Syntax:** `right(original, length)`

**Returns:**

A String.
Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>An expression that returns a string.</td>
</tr>
<tr>
<td>n</td>
<td>An expression that returns a positive integer.</td>
</tr>
</tbody>
</table>

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 345. Result

<table>
<thead>
<tr>
<th>right('hello', 3)</th>
<th>&quot;llo&quot;</th>
</tr>
</thead>
</table>

Rows: 1

10.8.6. rtrim()

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

Syntax: `rTrim(original)`

Returns:

A String.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>An expression that returns a string.</td>
</tr>
</tbody>
</table>

Considerations:

- `rTrim(null)` returns `null`

Query

```
RETURN rTrim('hello   ')
```
10.8.7. split()

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

Syntax: \texttt{split(original, splitDelimiter)}

Returns: A list of Strings.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>An expression that returns a string.</td>
</tr>
<tr>
<td>splitDelimiter</td>
<td>The string with which to split original.</td>
</tr>
</tbody>
</table>

Considerations:

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

Query

\texttt{RETURN split('one,two', ',')}
Arguments:

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

Considerations:

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

Query

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

Table 348. Result

<table>
<thead>
<tr>
<th>substring('hello', 1, 3)</th>
<th>substring('hello', 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;ell&quot;</td>
<td>&quot;llo&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

10.8.9. toLower()

toLower() returns the original string in lowercase.

Syntax: toLower(original)

Returns:

A String.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>An expression that returns a string.</td>
</tr>
</tbody>
</table>

Considerations:
toLower(null) returns null

Query

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

Table 349. Result

<table>
<thead>
<tr>
<th>toLower('HELLO')</th>
<th>&quot;hello&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

10.8.10. toString()

toString() converts an integer, float, boolean, string, point, duration, date, time, localtime, localdatetime or datetime value to a string.

Syntax: `toString(expression)`

Returns:

A String.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>An expression that returns a number, a boolean, string, temporal or spatial value.</td>
</tr>
</tbody>
</table>

Considerations:

<table>
<thead>
<tr>
<th>toString(null) returns null</th>
</tr>
</thead>
</table>

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

This function will return an error if provided with an expression that is not an integer, float, boolean, point, duration, date, time, localtime, localdatetime or datetime value.

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 350. Result
10.8.11. toStringOrNull()

The function `toStringOrNull()` converts an integer, float, boolean, string, point, duration, date, time, localtime, localdatetime or datetime value to a string.

**Syntax:** `toStringOrNull(expression)`

**Returns:**

A String or **null**.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>Any expression that returns a value.</td>
</tr>
</tbody>
</table>

**Considerations:**

`toStringOrNull(null)` returns **null**.

If the `expression` is not an integer, float, string, boolean, point, duration, date, time, localtime, localdatetime or datetime value, **null** will be returned.

**Query**

```sql
RETURN toStringOrNull(11.5),
toStringOrNull('already a string'),
toStringOrNull(true),
toStringOrNull(date({year:1984, month:10, day:11})) AS dateString,
toStringOrNull(datetime({year:1984, month:10, day:11, hour:12, minute:31, second:14, millisecond:341, timezone:'Europe/Stockholm'})) AS datetimeString,
toStringOrNull(duration({minutes:12, seconds:60})) AS durationString,
toStringOrNull(['A', 'B', 'C']) AS list
```

**Table 351. Result**

<table>
<thead>
<tr>
<th>toStringOrNull(11.5)</th>
<th>toStringOrNull('already a string')</th>
<th>toStringOrNull(true)</th>
<th>dateString</th>
<th>datetimeString</th>
<th>durationString</th>
<th>list</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;11.5&quot;</td>
<td>&quot;already a string&quot;</td>
<td>&quot;true&quot;</td>
<td>&quot;1984-10-11&quot;</td>
<td>&quot;1984-10-11T12:31:14.341+01:00[Europe/Stockholm]&quot;</td>
<td>&quot;PT1M&quot;</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

Rows: 1
10.8.12. toUpper()

**toUpper()** returns the original string in uppercase.

**Syntax:** `toUpper(original)`

**Returns:**

A String.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>An expression that returns a string.</td>
</tr>
</tbody>
</table>

**Considerations:**

`toUpper(null)` returns `null`

**Query**

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

**Table 352. Result**

<table>
<thead>
<tr>
<th><code>toUpper('hello')</code></th>
<th>&quot;HELLO&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

10.8.13. trim()

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

**Syntax:** `trim(original)`

**Returns:**

A String.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>An expression that returns a string.</td>
</tr>
</tbody>
</table>

**Considerations:**

`trim(null)` returns `null`
10.9. Temporal functions - instant types

Cypher provides functions allowing for the creation and manipulation of values for each temporal type — Date, Time, LocalTime, DateTime, and LocalDateTime.

See also Temporal (Date/Time) values and Temporal operators.

10.9.1. Temporal instant types

An introduction to temporal instant types, including descriptions of creation functions, clocks, and truncation.

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 354. Temporal instant type creation functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Date</th>
<th>Time</th>
<th>LocalTime</th>
<th>DateTime</th>
<th>LocalDateTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Getting the current value</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Creating a calendar-based (Year-Month-Day) value</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Creating a week-based (Year-Week-Day) value</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Creating a quarter-based (Year-Quarter-Day) value</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Creating an ordinal (Year-Day) value</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Creating a value from time components</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Creating a value from other temporal values using extractors (i.e. converting between different types)</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Creating a value from a string</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Creating a value from a timestamp</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Type</th>
<th>default</th>
<th>transaction</th>
<th>statement</th>
<th>realtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>date()</td>
<td>date.transaction()</td>
<td>date.statement()</td>
<td>date.realtime()</td>
</tr>
<tr>
<td>Time</td>
<td>time()</td>
<td>time.transaction()</td>
<td>time.statement()</td>
<td>time.realtime()</td>
</tr>
<tr>
<td>LocalTime</td>
<td>localtime()</td>
<td>localtime.transaction()</td>
<td>localtime.statement()</td>
<td>localtime.realtime()</td>
</tr>
<tr>
<td>DateTime</td>
<td>datetime()</td>
<td>datetime.transaction()</td>
<td>datetime.statement()</td>
<td>datetime.realtime()</td>
</tr>
<tr>
<td>LocalDateTime</td>
<td>localtime()</td>
<td>localtime.transaction()</td>
<td>localtime.statement()</td>
<td>localtime.realtime()</td>
</tr>
</tbody>
</table>

**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 millennium 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:

<table>
<thead>
<tr>
<th>Truncation unit</th>
<th>Date</th>
<th>Time</th>
<th>LocalTime</th>
<th>DateTime</th>
<th>LocalDateTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>millennium</td>
<td>date.truncate('millennium', input)</td>
<td></td>
<td></td>
<td>datetime.truncate('millennium', input)</td>
<td>localdatetime.truncate('millennium', input)</td>
</tr>
<tr>
<td>century</td>
<td>date.truncate('century', input)</td>
<td></td>
<td></td>
<td>datetime.truncate('century', input)</td>
<td>localdatetime.truncate('century', input)</td>
</tr>
<tr>
<td>decade</td>
<td>date.truncate('decade', input)</td>
<td></td>
<td></td>
<td>datetime.truncate('decade', input)</td>
<td>localdatetime.truncate('decade', input)</td>
</tr>
<tr>
<td>year</td>
<td>date.truncate('year', input)</td>
<td></td>
<td></td>
<td>datetime.truncate('year', input)</td>
<td>localdatetime.truncate('year', input)</td>
</tr>
<tr>
<td>weekYear</td>
<td>date.truncate('weekYear', input)</td>
<td></td>
<td></td>
<td>datetime.truncate('weekYear', input)</td>
<td>localdatetime.truncate('weekYear', input)</td>
</tr>
<tr>
<td>quarter</td>
<td>date.truncate('quarter', input)</td>
<td></td>
<td></td>
<td>datetime.truncate('quarter', input)</td>
<td>localdatetime.truncate('quarter', input)</td>
</tr>
<tr>
<td>month</td>
<td>date.truncate('month', input)</td>
<td></td>
<td></td>
<td>datetime.truncate('month', input)</td>
<td>localdatetime.truncate('month', input)</td>
</tr>
<tr>
<td>week</td>
<td>date.truncate('week', input)</td>
<td></td>
<td></td>
<td>datetime.truncate('week', input)</td>
<td>localdatetime.truncate('week', input)</td>
</tr>
<tr>
<td>day</td>
<td>date.truncate('day', input)</td>
<td>time.truncate('day', input)</td>
<td>localtime.truncate('day', input)</td>
<td>datetime.truncate('day', input)</td>
<td>localdatetime.truncate('day', input)</td>
</tr>
<tr>
<td>hour</td>
<td></td>
<td>time.truncate('hour', input)</td>
<td>localtime.truncate('hour', input)</td>
<td>datetime.truncate('hour', input)</td>
<td>localdatetime.truncate('hour', input)</td>
</tr>
<tr>
<td>minute</td>
<td></td>
<td>time.truncate('minute', input)</td>
<td>localtime.truncate('minute', input)</td>
<td>datetime.truncate('minute', input)</td>
<td>localdatetime.truncate('minute', input)</td>
</tr>
<tr>
<td>second</td>
<td></td>
<td>time.truncate('second', input)</td>
<td>localtime.truncate('second', input)</td>
<td>datetime.truncate('second', input)</td>
<td>localdatetime.truncate('second', input)</td>
</tr>
<tr>
<td>millisecond</td>
<td></td>
<td>time.truncate('millisecond', input)</td>
<td>localtime.truncate('millisecond', input)</td>
<td>datetime.truncate('millisecond', input)</td>
<td>localdatetime.truncate('millisecond', input)</td>
</tr>
<tr>
<td>microsecond</td>
<td></td>
<td>time.truncate('microsecond', input)</td>
<td>localtime.truncate('microsecond', input)</td>
<td>datetime.truncate('microsecond', input)</td>
<td>localdatetime.truncate('microsecond', input)</td>
</tr>
</tbody>
</table>

10.9.2. Date: **date()**

Details for using the **date()** function.
• Getting the current Date
• Creating a calendar (Year-Month-Day) Date
• Creating a week (Year-Week-Day) Date
• Creating a quarter (Year-Quarter-Day) Date
• Creating an ordinal (Year-Day) Date
• Creating a Date from a string
• Creating a Date using other temporal values as components
• Truncating a 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>timezone</td>
<td>A string expression that represents the time zone</td>
</tr>
</tbody>
</table>

Considerations:

If no parameters are provided, date() must be invoked (date() is invalid).

Query

```
RETURN date() AS currentDate
```

The current date is returned.

Table 355. Result

<table>
<thead>
<tr>
<th>currentDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>2022-08-09</td>
</tr>
</tbody>
</table>

Rows: 1
The current date in California is returned.

Table 356. Result

<table>
<thead>
<tr>
<th>currentDateInLA</th>
<th>2022-08-09</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

**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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timezone</td>
<td>A string expression that represents the <code>time zone</code></td>
</tr>
</tbody>
</table>

**Query**

```
RETURN date.transaction() AS currentDate
```

Table 357. Result

<table>
<thead>
<tr>
<th>currentDate</th>
<th>2022-08-09</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timezone</td>
<td>A string expression that represents the time zone</td>
</tr>
</tbody>
</table>

Query

```sql
RETURN date.statement() AS currentDate
```

Table 358. Result

<table>
<thead>
<tr>
<th>currentDate</th>
<th>2022-08-09</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows:</td>
<td>1</td>
</tr>
</tbody>
</table>

`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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timezone</td>
<td>A string expression that represents the time zone</td>
</tr>
</tbody>
</table>

Query

```sql
RETURN date.realtime() AS currentDate
```

Table 359. Result

<table>
<thead>
<tr>
<th>currentDate</th>
<th>2022-08-09</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows:</td>
<td>1</td>
</tr>
</tbody>
</table>
Query

```csharp
RETURN date.realtime('America/Los Angeles') AS currentDateInLA
```

Table 360. Result

<table>
<thead>
<tr>
<th>currentDateInLA</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2022-08-09</td>
<td></td>
</tr>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>year</td>
<td>An expression consisting of at least four digits that specifies the year.</td>
</tr>
<tr>
<td>month</td>
<td>An integer between 1 and 12 that specifies the month.</td>
</tr>
<tr>
<td>day</td>
<td>An integer between 1 and 31 that specifies the day of the month.</td>
</tr>
</tbody>
</table>

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

```csharp
UNWIND [date({year:1984, month:10, day:11}),
        date({year:1984, month:10}),
        date({year:1984})] as theDate
RETURN theDate
```

Table 361. Result
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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>year</td>
<td>An expression consisting of at least four digits that specifies the year.</td>
</tr>
<tr>
<td>week</td>
<td>An integer between 1 and 53 that specifies the week.</td>
</tr>
<tr>
<td>dayOfWeek</td>
<td>An integer between 1 and 7 that specifies the day of the week.</td>
</tr>
</tbody>
</table>

**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
UNWIND [
    date({year:1984, week:10, dayOfWeek:3}),
    date({year:1984, week:10}),
    date({year:1984})
] as theDate
RETURN theDate
```

**Table 362. Result**

<table>
<thead>
<tr>
<th>theDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984-03-07</td>
</tr>
<tr>
<td>1984-03-05</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>year</td>
<td>An expression consisting of at least four digits that specifies the year.</td>
</tr>
<tr>
<td>quarter</td>
<td>An integer between 1 and 4 that specifies the quarter.</td>
</tr>
<tr>
<td>dayOfQuarter</td>
<td>An integer between 1 and 92 that specifies the day of the quarter.</td>
</tr>
</tbody>
</table>

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

```query
UNWIND [date({year:1984, quarter:3, dayOfQuarter: 45}),
    date({year:1984, quarter:3}),
    date({year:1984})
] as theDate
RETURN theDate
```

Table 363. Result

<table>
<thead>
<tr>
<th>theDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984-08-14</td>
</tr>
<tr>
<td>1984-07-01</td>
</tr>
<tr>
<td>1984-01-01</td>
</tr>
</tbody>
</table>

Rows: 3
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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>year</td>
<td>An expression consisting of at least four digits that specifies the year.</td>
</tr>
<tr>
<td>ordinalDay</td>
<td>An integer between 1 and 366 that specifies the ordinal day of the year.</td>
</tr>
</tbody>
</table>

Considerations:

The ordinal day of the year component will default to 1 if ordinalDay is omitted.

Query

```sql
UNWIND [date({year:1984, ordinalDay:202}), date({year:1984})] as theDate
RETURN theDate
```

The date corresponding to 11 February 1984 is returned.

Table 364. Result

<table>
<thead>
<tr>
<th>theDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984-07-20</td>
</tr>
<tr>
<td>1984-01-01</td>
</tr>
</tbody>
</table>

Rows: 2

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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>temporalValue</td>
<td>A string representing a temporal value.</td>
</tr>
</tbody>
</table>

Considerations:

- `temporalValue` must comply with the format defined for `dates`.
- `temporalValue` must denote a valid date; i.e. a `temporalValue` denoting 30 February 2001 is invalid.
- `date(null)` returns null.

Query

```
UNWIND [
  date('2015-07-21'),
  date('2015-07'),
  date('201507'),
  date('2015-W30-2'),
  date('2015202'),
  date('2015')
] as theDate
RETURN theDate
```

Table 365. Result

<table>
<thead>
<tr>
<th>theDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015-07-21</td>
</tr>
<tr>
<td>2015-07-01</td>
</tr>
<tr>
<td>2015-07-01</td>
</tr>
<tr>
<td>2015-07-21</td>
</tr>
<tr>
<td>2015-07-21</td>
</tr>
<tr>
<td>2015-01-01</td>
</tr>
</tbody>
</table>

Rows: 6

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:
A single map consisting of the following:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>A Date value.</td>
</tr>
<tr>
<td>year</td>
<td>An expression consisting of at least four digits that specifies the year.</td>
</tr>
<tr>
<td>month</td>
<td>An integer between 1 and 12 that specifies the month.</td>
</tr>
<tr>
<td>day</td>
<td>An integer between 1 and 31 that specifies the day of the month.</td>
</tr>
<tr>
<td>week</td>
<td>An integer between 1 and 53 that specifies the week.</td>
</tr>
<tr>
<td>dayOfWeek</td>
<td>An integer between 1 and 7 that specifies the day of the week.</td>
</tr>
<tr>
<td>quarter</td>
<td>An integer between 1 and 4 that specifies the quarter.</td>
</tr>
<tr>
<td>dayOfQuarter</td>
<td>An integer between 1 and 92 that specifies the day of the quarter.</td>
</tr>
<tr>
<td>ordinalDay</td>
<td>An integer between 1 and 366 that specifies the ordinal day of the year.</td>
</tr>
</tbody>
</table>

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 366. Result

<table>
<thead>
<tr>
<th>dateOnly</th>
<th>dateDay</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984-11-11</td>
<td>1984-11-28</td>
</tr>
<tr>
<td>1984-11-11</td>
<td>1984-11-28</td>
</tr>
<tr>
<td>1984-11-11</td>
<td>1984-11-28</td>
</tr>
</tbody>
</table>

Rows: 3

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: \(\text{date.truncate(unit [, temporalInstantValue [, mapOfComponents ] ])}\)

Returns:
A Date.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unit</td>
<td>A string expression evaluating to one of the following: {millennium, century, decade, year, weekYear, quarter, month, week, day}.</td>
</tr>
<tr>
<td>temporalInstantValue</td>
<td>An expression of one of the following types: {DateTime, LocalDateTime, Date}.</td>
</tr>
<tr>
<td>mapOfComponents</td>
<td>An expression evaluating to a map containing components less significant than unit.</td>
</tr>
</tbody>
</table>

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.

If temporalInstantValue is not provided, it will be set to the current date, i.e. date.truncate(unit) is equivalent of date.truncate(unit, date()).

Query

```sql
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 367. Result
10.9.3. DateTime: **datetime()**

Details for using the **datetime()** function.

- Getting the current DateTime
- Creating a calendar (Year-Month-Day) DateTime
- Creating a week (Year-Week-Day) DateTime
- Creating a quarter (Year-Quarter-Day) DateTime
- Creating an ordinal (Year-Day) DateTime
- Creating a DateTime from a string
- Creating a DateTime using other temporal values as components
- Creating a DateTime from a timestamp
- Truncating a 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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>timezone</td>
<td>A string expression that represents the time zone</td>
</tr>
</tbody>
</table>

**Considerations:**

If no parameters are provided, **datetime()** must be invoked (**datetime()** is invalid).

**Query**

```sql
RETURN datetime() AS currentDateTime
```
The current date and time using the local time zone is returned.

Table 368. Result

<table>
<thead>
<tr>
<th>currentDateTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>2022-08-09T13:06:52.476Z</td>
</tr>
</tbody>
</table>

Rows: 1

Query

```
RETURN datetime({'timezone': 'America/Los Angeles'}) AS currentDateTimeInLA
```

The current date and time of day in California is returned.

Table 369. Result

<table>
<thead>
<tr>
<th>currentDateTimeInLA</th>
</tr>
</thead>
<tbody>
<tr>
<td>2022-08-09T06:06:52.491-07:00[America/Los_Angeles]</td>
</tr>
</tbody>
</table>

Rows: 1

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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timezone</td>
<td>A string expression that represents the time zone</td>
</tr>
</tbody>
</table>

Query

```
RETURN datetime.transaction() AS currentDateTime
```

Table 370. Result

<table>
<thead>
<tr>
<th>currentDateTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>2022-08-09T13:06:52.508Z</td>
</tr>
</tbody>
</table>

Rows: 1
Query

```
RETURN datetime.transaction('America/Los Angeles') AS currentDateTimeInLA
```

Table 371. Result

<table>
<thead>
<tr>
<th>currentDateTimeInLA</th>
</tr>
</thead>
<tbody>
<tr>
<td>2022-08-09T06:06:52.524-07:00[America/Los_Angeles]</td>
</tr>
</tbody>
</table>

Rows: 1

**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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timezone</td>
<td>A string expression that represents the time zone</td>
</tr>
</tbody>
</table>

Query

```
RETURN datetime.statement() AS currentDateTime
```

Table 372. Result

<table>
<thead>
<tr>
<th>currentDateTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>2022-08-09T13:06:52.545Z</td>
</tr>
</tbody>
</table>

Rows: 1

**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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timezone</td>
<td>A string expression that represents the timezone</td>
</tr>
</tbody>
</table>

**Query**

```
RETURN datetime().realtime() AS currentDateTime
```

**Table 373. Result**

<table>
<thead>
<tr>
<th>currentDateTime</th>
<th>2022-08-09T13:06:52.577240Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows:</td>
<td>1</td>
</tr>
</tbody>
</table>

**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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>year</td>
<td>An expression consisting of at least four digits that specifies the year.</td>
</tr>
<tr>
<td>month</td>
<td>An integer between 1 and 12 that specifies the month.</td>
</tr>
<tr>
<td>day</td>
<td>An integer between 1 and 31 that specifies the day of the month.</td>
</tr>
<tr>
<td>hour</td>
<td>An integer between 0 and 23 that specifies the hour of the day.</td>
</tr>
<tr>
<td>minute</td>
<td>An integer between 0 and 59 that specifies the number of minutes.</td>
</tr>
<tr>
<td>second</td>
<td>An integer between 0 and 59 that specifies the number of seconds.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>millisecond</td>
<td>An integer between 0 and 999 that specifies the number of milliseconds.</td>
</tr>
<tr>
<td>microsecond</td>
<td>An integer between 0 and 999,999 that specifies the number of microseconds.</td>
</tr>
<tr>
<td>nanosecond</td>
<td>An integer between 0 and 999,999,999 that specifies the number of nanoseconds.</td>
</tr>
<tr>
<td>timezone</td>
<td>An expression that specifies the time zone.</td>
</tr>
</tbody>
</table>

**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 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: 645, timezone: '+01:00'}),  
  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, timezone: 'Europe/Stockholm'})  
] as theDate  
RETURN theDate
```

**Table 374. Result**

<table>
<thead>
<tr>
<th>theDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984-10-11T12:31:14.645+01:00</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>year</td>
<td>An expression consisting of at least four digits that specifies the year.</td>
</tr>
<tr>
<td>week</td>
<td>An integer between 1 and 53 that specifies the week.</td>
</tr>
<tr>
<td>dayOfWeek</td>
<td>An integer between 1 and 7 that specifies the day of the week.</td>
</tr>
<tr>
<td>hour</td>
<td>An integer between 0 and 23 that specifies the hour of the day.</td>
</tr>
<tr>
<td>minute</td>
<td>An integer between 0 and 59 that specifies the number of minutes.</td>
</tr>
<tr>
<td>second</td>
<td>An integer between 0 and 59 that specifies the number of seconds.</td>
</tr>
<tr>
<td>millisecond</td>
<td>An integer between 0 and 999 that specifies the number of milliseconds.</td>
</tr>
<tr>
<td>microsecond</td>
<td>An integer between 0 and 999,999 that specifies the number of microseconds.</td>
</tr>
<tr>
<td>nanosecond</td>
<td>An integer between 0 and 999,999,999 that specifies the number of nanoseconds.</td>
</tr>
<tr>
<td>timezone</td>
<td>An expression that specifies the time zone.</td>
</tr>
</tbody>
</table>
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

```sql
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 375. Result

<table>
<thead>
<tr>
<th>theDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984-03-07T12:31:14.645Z</td>
</tr>
<tr>
<td>1984-03-07T12:31:14.645876+01:00</td>
</tr>
<tr>
<td>1984-03-07T12:31:14.645876123+01:00[Europe/Stockholm]</td>
</tr>
<tr>
<td>1984-03-07T12:31:14+01:00[Europe/Stockholm]</td>
</tr>
<tr>
<td>1984-03-07T12:31:14Z</td>
</tr>
<tr>
<td>1984-03-07T12:00+01:00</td>
</tr>
<tr>
<td>1984-03-07T00:00+01:00[Europe/Stockholm]</td>
</tr>
</tbody>
</table>

Rows: 7

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 DateTime.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>year</td>
<td>An expression consisting of at least four digits that specifies the year.</td>
</tr>
<tr>
<td>quarter</td>
<td>An integer between 1 and 4 that specifies the quarter.</td>
</tr>
<tr>
<td>dayOfQuarter</td>
<td>An integer between 1 and 92 that specifies the day of the quarter.</td>
</tr>
<tr>
<td>hour</td>
<td>An integer between 0 and 23 that specifies the hour of the day.</td>
</tr>
<tr>
<td>minute</td>
<td>An integer between 0 and 59 that specifies the number of minutes.</td>
</tr>
<tr>
<td>second</td>
<td>An integer between 0 and 59 that specifies the number of seconds.</td>
</tr>
<tr>
<td>millisecond</td>
<td>An integer between 0 and 999 that specifies the number of milliseconds.</td>
</tr>
<tr>
<td>microsecond</td>
<td>An integer between 0 and 999,999 that specifies the number of microseconds.</td>
</tr>
<tr>
<td>nanosecond</td>
<td>An integer between 0 and 999,999,999 that specifies the number of nanoseconds.</td>
</tr>
<tr>
<td>timezone</td>
<td>An expression that specifies the time zone.</td>
</tr>
</tbody>
</table>

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.
- 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, millisecond: 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 376. Result

<table>
<thead>
<tr>
<th>theDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984-08-14T12:31:14.645876Z</td>
</tr>
<tr>
<td>1984-08-14T12:31:14+01:00</td>
</tr>
<tr>
<td>1984-08-14T12:00+02:00[Europe/Stockholm]</td>
</tr>
<tr>
<td>1984-08-14T00:00Z</td>
</tr>
</tbody>
</table>

Rows: 4

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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>year</td>
<td>An expression consisting of at least four digits that specifies the year.</td>
</tr>
<tr>
<td>ordinalDay</td>
<td>An integer between 1 and 366 that specifies the ordinal day of the year.</td>
</tr>
<tr>
<td>hour</td>
<td>An integer between 0 and 23 that specifies the hour of the day.</td>
</tr>
<tr>
<td>minute</td>
<td>An integer between 0 and 59 that specifies the number of minutes.</td>
</tr>
</tbody>
</table>
### Name | Description
--- | ---
**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 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**

```sql
datetime({year:1984, ordinalDay:202, hour:12, minute:31, second:14, timezone: '+01:00'}),
datetime({year:1984, ordinalDay:202})] as theDate
RETURN theDate
```

---

**Table 377. Result**

<table>
<thead>
<tr>
<th>theDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984-07-20T12:31:14+01:00</td>
</tr>
<tr>
<td>1984-07-20T00:00+02:00[Europe/Stockholm]</td>
</tr>
<tr>
<td>1984-07-20T00:00Z</td>
</tr>
</tbody>
</table>

Rows: 4
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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>temporalValue</td>
<td>A string representing a temporal value.</td>
</tr>
</tbody>
</table>

Considerations:

- temporalValue must comply with the format defined for dates, times and time zones.
- 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.
- datetime(null) returns null.

Query

```
UNWIND ['2015-07-21T21:40:32.142+0100',
       '2015-W30-2T214032.142Z',
       '2015T214032-0100',
       '20150721T2140-0130',
       '2015-W30T2140-02',
       '2015202T21+18:00',
       '2015-07-21T21:40:32.142[Europe/London]',
       '2015-07-21T21:40:32.142-04[America/New_York]']
AS theDate
RETURN theDate
```

Table 378. Result

<table>
<thead>
<tr>
<th>theDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015-07-21T21:40:32.142+01:00</td>
</tr>
<tr>
<td>2015-07-21T21:40:32.142Z</td>
</tr>
<tr>
<td>2015-01-01T21:40:32-01:00</td>
</tr>
<tr>
<td>2015-07-21T21:40-01:30</td>
</tr>
<tr>
<td>2015-07-20T21:40-02:00</td>
</tr>
<tr>
<td>2015-07-21T21:00+18:00</td>
</tr>
<tr>
<td>2015-07-21T21:40:32.142+01:00[Europe/London]</td>
</tr>
</tbody>
</table>

Rows: 8
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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>datetime</td>
<td>A DateTime value.</td>
</tr>
<tr>
<td>date</td>
<td>A Date value.</td>
</tr>
<tr>
<td>time</td>
<td>A Time value.</td>
</tr>
<tr>
<td>year</td>
<td>An expression consisting of at least four digits that specifies the year.</td>
</tr>
<tr>
<td>month</td>
<td>An integer between 1 and 12 that specifies the month.</td>
</tr>
<tr>
<td>day</td>
<td>An integer between 1 and 31 that specifies the day of the month.</td>
</tr>
<tr>
<td>week</td>
<td>An integer between 1 and 53 that specifies the week.</td>
</tr>
<tr>
<td>dayOfWeek</td>
<td>An integer between 1 and 7 that specifies the day of the week.</td>
</tr>
<tr>
<td>quarter</td>
<td>An integer between 1 and 4 that specifies the quarter.</td>
</tr>
<tr>
<td>dayOfQuarter</td>
<td>An integer between 1 and 92 that specifies the day of the quarter.</td>
</tr>
<tr>
<td>ordinalDay</td>
<td>An integer between 1 and 366 that specifies the ordinal day of the year.</td>
</tr>
<tr>
<td>hour</td>
<td>An integer between 0 and 23 that specifies the hour of the day.</td>
</tr>
<tr>
<td>minute</td>
<td>An integer between 0 and 59 that specifies the number of minutes.</td>
</tr>
<tr>
<td>second</td>
<td>An integer between 0 and 59 that specifies the number of seconds.</td>
</tr>
<tr>
<td>millisecond</td>
<td>An integer between 0 and 999 that specifies the number of milliseconds.</td>
</tr>
<tr>
<td>microsecond</td>
<td>An integer between 0 and 999,999 that specifies the number of microseconds.</td>
</tr>
</tbody>
</table>
### Name | Description
---|---
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**

```sql
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 379. Result

<table>
<thead>
<tr>
<th>dateHHMMSS</th>
<th>dateHHMMSSTimezone</th>
<th>dateDDHHMMSS</th>
<th>dateDDHHMMSSTimezone</th>
</tr>
</thead>
</table>

Rows: 1

The following query shows the various usages of `datetime({time [, year, …, timezone]})`

**Query**

```sql
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 YYYYMMDDTimeSS,
    datetime((year:1984, month:10, day:11, time:tt, second: 42)) AS YYYYMMDDTimeZone,
    datetime((year:1984, month:10, day:11, time:tt, second: 42, timezone:'Pacific/Honolulu')) AS YYYYMMDDTimeSSTimeZone
```

### Table 380. Result
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 381. Result

<table>
<thead>
<tr>
<th>dateTime</th>
<th>dateTimeTimezone</th>
<th>dateTimeDDSS</th>
<th>dateTimeDDSSTimezone</th>
</tr>
</thead>
</table>

Rows: 1

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 382. Result

<table>
<thead>
<tr>
<th>dateTime</th>
<th>dateTimeTimezone</th>
<th>dateTimeDDSS</th>
<th>dateTimeDDSSTimezone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984-10-11T12:00+01:00[Europe/Stockholm]</td>
<td>1984-10-11T16:00+05:00</td>
<td>1984-10-28T12:00+42:00[Europe/Stockholm]</td>
<td>1984-10-28T01:00:42-10:00[Pacific/Honolulu]</td>
</tr>
</tbody>
</table>

Rows: 1

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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EpochSeconds</td>
<td>A numeric value representing the number of seconds from the UNIX epoch in the UTC time zone.</td>
</tr>
<tr>
<td>EpochMillis</td>
<td>A numeric value representing the number of milliseconds from the UNIX epoch in the UTC time zone.</td>
</tr>
</tbody>
</table>

Considerations:

**EpochSeconds/EpochMillis** may be used in conjunction with **nanosecond**

Query

```sql
RETURN datetime({ epochSeconds: timestamp() // 1000, nanosecond: 23 }) AS theDate
```

Table 383. Result

<table>
<thead>
<tr>
<th>theDate</th>
<th>2022-08-09T13:06:53.000000023Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows:</td>
<td>1</td>
</tr>
</tbody>
</table>

Query

```sql
RETURN datetime({ epochMillis: 424797300000 }) AS theDate
```

Table 384. Result

<table>
<thead>
<tr>
<th>theDate</th>
<th>1983-06-18T15:15Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows:</td>
<td>1</td>
</tr>
</tbody>
</table>

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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unit</td>
<td>A string expression evaluating to one of the following: millenium, century, decade, year, weekYear, quarter, month, week, day, hour, minute, second, millisecond, microsecond.</td>
</tr>
<tr>
<td>temporalInstantValue</td>
<td>An expression of one of the following types: {DateTime, LocalDateTime, Date}.</td>
</tr>
<tr>
<td>mapOfComponents</td>
<td>An expression evaluating to a map containing components less significant than unit. During truncation, a time zone can be attached or overridden using the key timezone.</td>
</tr>
</tbody>
</table>

**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.
- If `temporalInstantValue` is not provided, it will be set to the current date, time and timezone, i.e. `datetime.truncate(unit)` is equivalent of `datetime.truncate(unit, datetime())`. 
Query

```sql
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) 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 385. Result

<table>
<thead>
<tr>
<th>truncMillenium</th>
<th>truncYear</th>
<th>truncMonth</th>
<th>truncDay</th>
<th>truncHour</th>
<th>truncSecond</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01T00:00+01:00[Europe/Stockholm]</td>
<td>2017-01-05T00:00+03:00</td>
<td>2017-11-01T00:00+03:00</td>
<td>2017-11-11T00:00:00.002+03:00</td>
<td>2017-11-11T12:00+03:00</td>
<td>2017-11-11T12:31:14+03:00</td>
</tr>
</tbody>
</table>

Rows: 1

10.9.4. LocalDateTime: `localdatetime()`

Details for using the `localdatetime()` function.

- Getting the current LocalDateTime
- Creating a calendar (Year-Month-Day) LocalDateTime
- Creating a week (Year-Week-Day) LocalDateTime
- Creating a quarter (Year-Quarter-Day) LocalDateTime
- Creating an ordinal (Year-Day) LocalDateTime
- Creating a LocalDateTime from a string
- Creating a LocalDateTime using other temporal values as components
- Truncating a 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>timezone</td>
<td>A string expression that represents the time zone</td>
</tr>
</tbody>
</table>

Considerations:

If no parameters are provided, `localdatetime()` must be invoked `(localdatetime({}))` is invalid).

Query

```sql
RETURN localdatetime() AS now
```

The current local date and time (i.e. in the local time zone) is returned.

Table 386. Result

<table>
<thead>
<tr>
<th>now</th>
</tr>
</thead>
<tbody>
<tr>
<td>2022-08-09T13:06:53.349</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

Query

```sql
RETURN localdatetime({timezone: 'America/Los Angeles'}) AS now
```

The current local date and time in California is returned.

Table 387. Result

<table>
<thead>
<tr>
<th>now</th>
</tr>
</thead>
<tbody>
<tr>
<td>2022-08-09T06:06:53.362</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

`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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timezone</td>
<td>A string expression that represents the time zone</td>
</tr>
</tbody>
</table>
localdatetime.transaction()

`localdatetime.transaction()` 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.transaction()`

**Returns:**

A `LocalDateTime`.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timezone</td>
<td>A string expression that represents the time zone</td>
</tr>
</tbody>
</table>

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()`

**Returns:**
A `LocalDateTime`.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timezone</td>
<td>A string expression that represents the timezone</td>
</tr>
</tbody>
</table>

Query

```
RETURN localdatetime.realtime() AS now
```

Table 390. Result

<table>
<thead>
<tr>
<th>now</th>
</tr>
</thead>
<tbody>
<tr>
<td>2022-08-09T13:06:53.417895</td>
</tr>
</tbody>
</table>

Rows: 1

Query

```
RETURN localdatetime.realtime('America/Los Angeles') AS nowInLA
```

Table 391. Result

<table>
<thead>
<tr>
<th>nowInLA</th>
</tr>
</thead>
<tbody>
<tr>
<td>2022-08-09T06:06:53.432235</td>
</tr>
</tbody>
</table>

Rows: 1

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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>year</td>
<td>An expression consisting of at \textit{least four digits} that specifies</td>
</tr>
<tr>
<td></td>
<td>the year.</td>
</tr>
<tr>
<td>month</td>
<td>An integer between 1 and 12 that specifies the month.</td>
</tr>
</tbody>
</table>
### Name | Description
--- | ---
**day** | An integer between 1 and 31 that specifies the day of the month.

**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.

### 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.

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

```sql
RETURN localdatetime({year: 1984, month: 10, day: 11, hour: 12, minute: 31, second: 14, millisecond: 123, microsecond: 456, nanosecond: 789}) AS theDate
```

### Table 392. Result

<table>
<thead>
<tr>
<th>theDate</th>
</tr>
</thead>
</table>

Rows: 1
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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>year</td>
<td>An expression consisting of at least four digits that specifies the year.</td>
</tr>
<tr>
<td>week</td>
<td>An integer between 1 and 53 that specifies the week.</td>
</tr>
<tr>
<td>dayOfWeek</td>
<td>An integer between 1 and 7 that specifies the day of the week.</td>
</tr>
<tr>
<td>hour</td>
<td>An integer between 0 and 23 that specifies the hour of the day.</td>
</tr>
<tr>
<td>minute</td>
<td>An integer between 0 and 59 that specifies the number of minutes.</td>
</tr>
<tr>
<td>second</td>
<td>An integer between 0 and 59 that specifies the number of seconds.</td>
</tr>
<tr>
<td>millisecond</td>
<td>An integer between 0 and 999 that specifies the number of milliseconds.</td>
</tr>
<tr>
<td>microsecond</td>
<td>An integer between 0 and 999,999 that specifies the number of microseconds.</td>
</tr>
<tr>
<td>nanosecond</td>
<td>An integer between 0 and 999,999,999 that specifies the number of nanoseconds.</td>
</tr>
</tbody>
</table>

**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 393. Result

<table>
<thead>
<tr>
<th>theDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984-03-07T12:31:14.645</td>
</tr>
</tbody>
</table>

Rows: 1

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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>year</td>
<td>An expression consisting of at least four digits that specifies the year.</td>
</tr>
<tr>
<td>quarter</td>
<td>An integer between 1 and 4 that specifies the quarter.</td>
</tr>
<tr>
<td>dayOfQuarter</td>
<td>An integer between 1 and 92 that specifies the day of the quarter.</td>
</tr>
<tr>
<td>hour</td>
<td>An integer between 0 and 23 that specifies the hour of the day.</td>
</tr>
<tr>
<td>minute</td>
<td>An integer between 0 and 59 that specifies the number of minutes.</td>
</tr>
<tr>
<td>second</td>
<td>An integer between 0 and 59 that specifies the number of seconds.</td>
</tr>
</tbody>
</table>
### Name Description

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>millisecond</td>
<td>An integer between 0 and 999 that specifies the number of milliseconds.</td>
</tr>
<tr>
<td>microsecond</td>
<td>An integer between 0 and 999,999 that specifies the number of microseconds.</td>
</tr>
<tr>
<td>nanosecond</td>
<td>An integer between 0 and 999,999,999 that specifies the number of nanoseconds.</td>
</tr>
</tbody>
</table>

### 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 394. Result

<table>
<thead>
<tr>
<th>theDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984-08-14T12:31:14.645876123</td>
</tr>
</tbody>
</table>

Rows: 1

### 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>An expression consisting of at least four digits that specifies the year.</td>
</tr>
<tr>
<td>ordinalDay</td>
<td>An integer between 1 and 366 that specifies the ordinal day of the year.</td>
</tr>
<tr>
<td>hour</td>
<td>An integer between 0 and 23 that specifies the hour of the day.</td>
</tr>
<tr>
<td>minute</td>
<td>An integer between 0 and 59 that specifies the number of minutes.</td>
</tr>
<tr>
<td>second</td>
<td>An integer between 0 and 59 that specifies the number of seconds.</td>
</tr>
<tr>
<td>millisecond</td>
<td>An integer between 0 and 999 that specifies the number of milliseconds.</td>
</tr>
<tr>
<td>microsecond</td>
<td>An integer between 0 and 999,999 that specifies the number of microseconds.</td>
</tr>
<tr>
<td>nanosecond</td>
<td>An integer between 0 and 999,999,999 that specifies the number of nanoseconds.</td>
</tr>
</tbody>
</table>

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

```sql
RETURN localdatetime({year: 1984, ordinalDay: 202, hour: 12, minute: 31, second: 14, microsecond: 645876}) AS theDate
```

Table 395. Result

<table>
<thead>
<tr>
<th>theDate</th>
</tr>
</thead>
</table>
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 LocalDateTime.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>temporalValue</td>
<td>A string representing a temporal value.</td>
</tr>
</tbody>
</table>

**Considerations:**

- `temporalValue` must comply with the format defined for dates and times.
- `temporalValue` must denote a valid date and time; i.e. a `temporalValue` denoting 30 February 2001 is invalid.
- `localdatetime(null)` returns null.

**Query**

```sql
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 396. Result**

<table>
<thead>
<tr>
<th>theDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015-07-21T21:40:32.142</td>
</tr>
<tr>
<td>2015-07-21T21:40:32.142</td>
</tr>
<tr>
<td>2015-07-21T21:40:32</td>
</tr>
<tr>
<td>2015-07-21T21:00</td>
</tr>
</tbody>
</table>

Rows: 4
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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>datetime</td>
<td>A DateTime value.</td>
</tr>
<tr>
<td>date</td>
<td>A Date value.</td>
</tr>
<tr>
<td>time</td>
<td>A Time value.</td>
</tr>
<tr>
<td>year</td>
<td>An expression consisting of at least four digits that specifies the year.</td>
</tr>
<tr>
<td>month</td>
<td>An integer between 1 and 12 that specifies the month.</td>
</tr>
<tr>
<td>day</td>
<td>An integer between 1 and 31 that specifies the day of the month.</td>
</tr>
<tr>
<td>week</td>
<td>An integer between 1 and 53 that specifies the week.</td>
</tr>
<tr>
<td>dayOfWeek</td>
<td>An integer between 1 and 7 that specifies the day of the week.</td>
</tr>
<tr>
<td>quarter</td>
<td>An integer between 1 and 4 that specifies the quarter.</td>
</tr>
<tr>
<td>dayOfQuarter</td>
<td>An integer between 1 and 92 that specifies the day of the quarter.</td>
</tr>
<tr>
<td>ordinalDay</td>
<td>An integer between 1 and 366 that specifies the ordinal day of the year.</td>
</tr>
<tr>
<td>hour</td>
<td>An integer between 0 and 23 that specifies the hour of the day.</td>
</tr>
<tr>
<td>minute</td>
<td>An integer between 0 and 59 that specifies the number of minutes.</td>
</tr>
<tr>
<td>second</td>
<td>An integer between 0 and 59 that specifies the number of seconds.</td>
</tr>
<tr>
<td>millisecond</td>
<td>An integer between 0 and 999 that specifies the number of milliseconds.</td>
</tr>
</tbody>
</table>
### 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.

**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 dateHMMSS,
       localdatetime({date: dd, day: 28, hour: 10, minute: 10, second: 10}) AS dateDDHMMSS
```

**Table 397. Result**

<table>
<thead>
<tr>
<th>dateHMMSS</th>
<th>dateDDHMMSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984-10-11T10:10</td>
<td>1984-10-28T10:10</td>
</tr>
</tbody>
</table>

Rows: 1

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 398. Result**

<table>
<thead>
<tr>
<th>YYYYMMDDTime</th>
<th>YYYYMMDDTimeSS</th>
</tr>
</thead>
</table>

Rows: 1

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 399. Result**

<table>
<thead>
<tr>
<th>dateTime</th>
<th>dateTimeDDSS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Rows: 1
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 400. Result**

<table>
<thead>
<tr>
<th>dateTime</th>
<th>dateTimeDDSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984-10-11T12:00</td>
<td>1984-10-28T12:00:42</td>
</tr>
</tbody>
</table>

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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unit</td>
<td>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>.</td>
</tr>
<tr>
<td>temporalInstantValue</td>
<td>An expression of one of the following types: <code>[DateTime, LocalDateTime, Date]</code>.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>mapOfComponents</td>
<td>An expression evaluating to a map containing components less significant than unit.</td>
</tr>
</tbody>
</table>

**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.
- If `temporalInstantValue` is not provided, it will be set to the current date and time, i.e. `localdatetime.truncate(unit)` is equivalent of `localdatetime.truncate(unit, localdatetime())`.

**Query**

```sql
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) 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 401. Result**

<table>
<thead>
<tr>
<th>truncMillenium</th>
<th>truncYear</th>
<th>truncMonth</th>
<th>truncDay</th>
<th>truncHour</th>
<th>truncSecond</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01T00:00</td>
<td>2017-01-02T00:00</td>
<td>2017-11-01T00:00</td>
<td>2017-11-11T00:00</td>
<td>2017-11-11T12:00:00.0000000002</td>
<td>2017-11-11T12:31:14</td>
</tr>
</tbody>
</table>

Rows: 1

### 10.9.5. LocalTime: `localtime()`

*Details for using the `localtime()` function.*

- Getting the current LocalTime
- Creating a LocalTime
- Creating a LocalTime from a string
- Creating a LocalTime using other temporal values as components
- Truncating a 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>timezone</td>
<td>A string expression that represents the time zone</td>
</tr>
</tbody>
</table>

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 402. Result

<table>
<thead>
<tr>
<th>now</th>
</tr>
</thead>
<tbody>
<tr>
<td>13:06:53.792</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

Query

```
RETURN localtime([timezone: 'America/Los Angeles']) AS nowInLA
```

The current local time in California is returned.

Table 403. Result

<table>
<thead>
<tr>
<th>nowInLA</th>
</tr>
</thead>
<tbody>
<tr>
<td>06:06:53.808</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timezone</td>
<td>A string expression that represents the time zone</td>
</tr>
</tbody>
</table>

Query

```
RETURN localtime.transaction() AS now
```

Table 404. Result

<table>
<thead>
<tr>
<th>now</th>
<th>13:06:53.823</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows:</td>
<td>1</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timezone</td>
<td>A string expression that represents the time zone</td>
</tr>
</tbody>
</table>

Query

```
RETURN localtime.statement() AS now
```
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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timezone</td>
<td>A string expression that represents the time zone</td>
</tr>
</tbody>
</table>

**Query**

```
RETURN localtime.realtime() AS now
```

Table 407. Result

<table>
<thead>
<tr>
<th>now</th>
</tr>
</thead>
<tbody>
<tr>
<td>13:06:53.878097</td>
</tr>
</tbody>
</table>

Rows: 1

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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hour</td>
<td>An integer between 0 and 23 that specifies the hour of the day.</td>
</tr>
<tr>
<td>minute</td>
<td>An integer between 0 and 59 that specifies the number of minutes.</td>
</tr>
<tr>
<td>second</td>
<td>An integer between 0 and 59 that specifies the number of seconds.</td>
</tr>
<tr>
<td>millisecond</td>
<td>An integer between 0 and 999 that specifies the number of milliseconds.</td>
</tr>
<tr>
<td>microsecond</td>
<td>An integer between 0 and 999,999 that specifies the number of microseconds.</td>
</tr>
<tr>
<td>nanosecond</td>
<td>An integer between 0 and 999,999,999 that specifies the number of nanoseconds.</td>
</tr>
</tbody>
</table>

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.
- 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

```sql
localtime({hour:12, minute:31, second:14}),
localtime({hour:12})] as theTime
RETURN theTime
```

Table 408. Result
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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>temporalValue</code></td>
<td>A string representing a temporal value.</td>
</tr>
</tbody>
</table>

**Considerations:**

- `temporalValue` must comply with the format defined for `times`.
- `temporalValue` must denote a valid time; i.e. a `temporalValue` denoting `13:46:64` is invalid.
- `localtime(null)` returns null.

**Query**

```
UNWIND [localtime('21:40:32.142'),
       localtime('214032.142'),
       localtime('21:40'),
       localtime('21')]
AS theTime
RETURN theTime
```

**Table 409. Result**

<table>
<thead>
<tr>
<th>theTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>21:40:32.142</td>
</tr>
<tr>
<td>21:40:32.142</td>
</tr>
<tr>
<td>21:40</td>
</tr>
<tr>
<td>21:00</td>
</tr>
</tbody>
</table>

Rows: 4
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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td><code>time</code></td>
<td>A Time value.</td>
</tr>
<tr>
<td><code>hour</code></td>
<td>An integer between 0 and 23 that specifies the hour of the day.</td>
</tr>
<tr>
<td><code>minute</code></td>
<td>An integer between 0 and 59 that specifies the number of minutes.</td>
</tr>
<tr>
<td><code>second</code></td>
<td>An integer between 0 and 59 that specifies the number of seconds.</td>
</tr>
<tr>
<td><code>millisecond</code></td>
<td>An integer between 0 and 999 that specifies the number of milliseconds.</td>
</tr>
<tr>
<td><code>microsecond</code></td>
<td>An integer between 0 and 999,999 that specifies the number of microseconds.</td>
</tr>
<tr>
<td><code>nanosecond</code></td>
<td>An integer between 0 and 999,999,999 that specifies the number of nanoseconds.</td>
</tr>
</tbody>
</table>

**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**

```sql
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 410. Result**

<table>
<thead>
<tr>
<th>timeOnly</th>
<th>timeSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:31:14.645876</td>
<td>12:31:42.645876</td>
</tr>
</tbody>
</table>

Rows: 1
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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unit</td>
<td>A string expression evaluating to one of the following: {day, hour, minute, second, millisecond, microsecond}.</td>
</tr>
<tr>
<td>temporalInstantValue</td>
<td>An expression of one of the following types: {DateTime, LocalDateTime, Time, LocalTime}.</td>
</tr>
<tr>
<td>mapOfComponents</td>
<td>An expression evaluating to a map containing components less significant than unit.</td>
</tr>
</tbody>
</table>

**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.

If temporalInstantValue is not provided, it will be set to the current time, i.e. `localtime.truncate(unit)` is equivalent of `localtime.truncate(unit, localtime())`. 
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 411. Result

<table>
<thead>
<tr>
<th>truncDay</th>
<th>truncHour</th>
<th>truncMinute</th>
<th>truncSecond</th>
<th>truncMillisecond</th>
<th>truncMicrosecond</th>
</tr>
</thead>
</table>

Rows: 1

10.9.6. Time: time()

Details for using the time() function.

- Getting the current Time
- Creating a Time
- Creating a Time from a string
- Creating a Time using other temporal values as components
- Truncating a 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({time zone})

Returns:

A Time.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>timezone</td>
<td>A string expression that represents the time zone</td>
</tr>
</tbody>
</table>

Considerations:

If no parameters are provided, time() must be invoked (time() is invalid).
The current time of day using the local time zone is returned.

Table 412. Result

<table>
<thead>
<tr>
<th>currentTime</th>
<th>13:06:54.037Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

The current time of day in California is returned.

Table 413. Result

<table>
<thead>
<tr>
<th>currentTimeInLA</th>
<th>06:06:54.051-07:00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timezone</td>
<td>A string expression that represents the time zone</td>
</tr>
</tbody>
</table>

Query

```csharp
RETURN time() AS currentTime
```

Table 414. Result
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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timezone</td>
<td>A string expression that represents the <strong>time zone</strong></td>
</tr>
</tbody>
</table>

**Query**

```
RETURN time.statement() AS currentTime
```

**Table 415. Result**

<table>
<thead>
<tr>
<th>currentTime</th>
<th>13:06:54.067Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows:</td>
<td>1</td>
</tr>
</tbody>
</table>

**Query**

```
RETURN time.statement('America/Los Angeles') AS currentTimeInLA
```

**Table 416. Result**

<table>
<thead>
<tr>
<th>currentTimeInLA</th>
<th>06:06:54.095-07:00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows:</td>
<td>1</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timezone</td>
<td>A string expression that represents the timezone</td>
</tr>
</tbody>
</table>

Query

```
RETURN time.realtime() AS currentTime
```

Table 417. Result

<table>
<thead>
<tr>
<th>currentTime</th>
<th>13:06:54.123476Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows:</td>
<td>1</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of</td>
<td>Description</td>
</tr>
<tr>
<td>the following:</td>
<td></td>
</tr>
<tr>
<td>hour</td>
<td>An integer between 0 and 23 that specifies the hour of the day.</td>
</tr>
<tr>
<td>minute</td>
<td>An integer between 0 and 59 that specifies the number of minutes.</td>
</tr>
<tr>
<td>second</td>
<td>An integer between 0 and 59 that specifies the number of seconds.</td>
</tr>
<tr>
<td>millisecond</td>
<td>An integer between 0 and 999 that specifies the number of milliseconds.</td>
</tr>
<tr>
<td>microsecond</td>
<td>An integer between 0 and 999,999 that specifies the number of microseconds.</td>
</tr>
</tbody>
</table>
Considerations:

- The hour component will default to 0 if hour is omitted.
- The second component will default to 0 if second is omitted.
- The timezone component will default to the configured default time zone if timezone 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 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

```sql
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, microsecond: 645876, timezone: '+01:00'}),
  time({hour: 12, timezone: '+01:00'})
] AS theTime
RETURN theTime
```

Table 418. Result

<table>
<thead>
<tr>
<th>theTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:31:14.123456789Z</td>
</tr>
<tr>
<td>12:31:14.645876123Z</td>
</tr>
<tr>
<td>12:31:14.645876+01:00</td>
</tr>
<tr>
<td>12:31+01:00</td>
</tr>
<tr>
<td>12:00+01:00</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>temporalValue</td>
<td>A string representing a temporal value.</td>
</tr>
</tbody>
</table>

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.
- `temporalValue` must denote a valid time; i.e. a `temporalValue` denoting 15:67 is invalid.
- `time(null)` returns null.

Query

```sql
UNWIND [
  time('21:40:32.142+01:00'),
  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 419. Result

<table>
<thead>
<tr>
<th>theTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>21:40:32.142+01:00</td>
</tr>
<tr>
<td>21:40:32.142Z</td>
</tr>
<tr>
<td>21:40:32+01:00</td>
</tr>
<tr>
<td>21:40:32-01:00</td>
</tr>
<tr>
<td>21:40-01:30</td>
</tr>
<tr>
<td>21:40-00</td>
</tr>
<tr>
<td>21:40-02</td>
</tr>
<tr>
<td>22:00+18:00</td>
</tr>
</tbody>
</table>

Rows: 8

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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>time</td>
<td>A Time value.</td>
</tr>
<tr>
<td>hour</td>
<td>An integer between 0 and 23 that specifies the hour of the day.</td>
</tr>
<tr>
<td>minute</td>
<td>An integer between 0 and 59 that specifies the number of minutes.</td>
</tr>
<tr>
<td>second</td>
<td>An integer between 0 and 59 that specifies the number of seconds.</td>
</tr>
<tr>
<td>millisecond</td>
<td>An integer between 0 and 999 that specifies the number of milliseconds.</td>
</tr>
<tr>
<td>microsecond</td>
<td>An integer between 0 and 999,999 that specifies the number of microseconds.</td>
</tr>
<tr>
<td>nanosecond</td>
<td>An integer between 0 and 999,999,999 that specifies the number of nanoseconds.</td>
</tr>
<tr>
<td>timezone</td>
<td>An expression that specifies the time zone.</td>
</tr>
</tbody>
</table>

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

```sql
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 timezone,
    time({time: tt, second: 42}) AS timeSS,
    time({time: tt, second: 42, timezone: '+05:00'}) AS timeSSTimezone
```

Table 420. Result

<table>
<thead>
<tr>
<th>timeOnly</th>
<th>timeZone</th>
<th>timeSS</th>
<th>timeSSTimezone</th>
</tr>
</thead>
</table>

Rows: 1
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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unit</td>
<td>A string expression evaluating to one of the following: {day, hour, minute, second, millisecond, microsecond}.</td>
</tr>
<tr>
<td>temporalInstantValue</td>
<td>An expression of one of the following types: {DateTime, LocalDateTime, Time, LocalTime}.</td>
</tr>
<tr>
<td>mapOfComponents</td>
<td>An expression evaluating to a map containing components less significant than unit. During truncation, a time zone can be attached or overridden using the key timezone.</td>
</tr>
</tbody>
</table>

Considerations:

<table>
<thead>
<tr>
<th>Consideration</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>The time zone of temporalInstantValue may be overridden; for example, time.truncate('minute', input, {timezone:'+0200'}).</td>
<td></td>
</tr>
<tr>
<td>If temporalInstantValue is one of {Time, DateTime} — a value with a time zone — and the time zone is overridden, no time conversion occurs.</td>
<td></td>
</tr>
<tr>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Any component that is not contained in mapOfComponents and which is less significant than unit will be set to its minimal value.</td>
<td></td>
</tr>
<tr>
<td>If mapOfComponents is not provided, all components of the returned value which are less significant than unit will be set to their default values.</td>
<td></td>
</tr>
</tbody>
</table>
If `temporalInstantValue` is not provided, it will be set to the current time and timezone, i.e. `time.truncate(unit)` is equivalent of `time.truncate(unit, time())`.

**Query**

```cypher
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 truncMilliseconds,
      time.truncate('microsecond', t) AS truncMicrosecond
```

**Table 421. Result**

<table>
<thead>
<tr>
<th>truncDay</th>
<th>truncHour</th>
<th>truncMinute</th>
<th>truncSecond</th>
<th>truncMilliseconds</th>
<th>truncMicrosecond</th>
</tr>
</thead>
<tbody>
<tr>
<td>00:00-01:00</td>
<td>12:00-01:00</td>
<td>12:31-01:00</td>
<td>12:31:14-01:00</td>
<td>12:31:14:645000000-01:00</td>
<td>12:31:14:645876-01:00</td>
</tr>
</tbody>
</table>

Rows: 1

10.10. Temporal functions - duration

*Cypher provides functions allowing for the creation and manipulation of values for a Duration temporal type.*

| See also Temporal (Date/Time) values and Temporal operators. |

**duration()**:

- Creating a Duration from duration components
- Creating a Duration from a string
- Computing the Duration between two temporal instants

Information regarding specifying and accessing components of a Duration value can be found [here](#).

10.10.1. 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>years</td>
<td>A numeric expression.</td>
</tr>
<tr>
<td>quarters</td>
<td>A numeric expression.</td>
</tr>
<tr>
<td>months</td>
<td>A numeric expression.</td>
</tr>
<tr>
<td>weeks</td>
<td>A numeric expression.</td>
</tr>
<tr>
<td>days</td>
<td>A numeric expression.</td>
</tr>
<tr>
<td>hours</td>
<td>A numeric expression.</td>
</tr>
<tr>
<td>minutes</td>
<td>A numeric expression.</td>
</tr>
<tr>
<td>seconds</td>
<td>A numeric expression.</td>
</tr>
<tr>
<td>milliseconds</td>
<td>A numeric expression.</td>
</tr>
<tr>
<td>microseconds</td>
<td>A numeric expression.</td>
</tr>
<tr>
<td>nanoseconds</td>
<td>A numeric expression.</td>
</tr>
</tbody>
</table>

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.
10.10.2. 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>temporalAmount</td>
<td>A string representing a temporal amount.</td>
</tr>
</tbody>
</table>

Considerations:

temporalAmount must comply with either the unit based form or date-and-time based form defined for Durations.

Query

```sql
UNWIND [
  duration("P14DT16H12M"),
  duration("P5M1.5D"),
  duration("P0.75M"),
  duration("PT0.75M"),
  duration("2012-02-02T14:37:21.545")
] AS aDuration
RETURN aDuration
```
10.10.3. 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(instant_1, instant_2)`

**Returns:**
A Duration.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>instant_1</td>
<td>An expression returning any temporal instant type (Date etc) that represents the starting instant.</td>
</tr>
<tr>
<td>instant_2</td>
<td>An expression returning any temporal instant type (Date etc) that represents the ending instant.</td>
</tr>
</tbody>
</table>

Considerations:
If `instant_1` occurs earlier than `instant_2`, the resulting `Duration` will be negative.

If `instant_1` has a time component and `instant_2` does not, the time component of `instant_1` is assumed to be midnight, and vice versa.

If `instant_1` has a time zone component and `instant_2` does not, the time zone component of `instant_1` is assumed to be the same as that of `instant_2`, and vice versa.

If `instant_1` has a date component and `instant_2` does not, the date component of `instant_2` is assumed to be the same as that of `instant_1`, and vice versa.

**Query**

```sql
UNWIND [duration.between(date("1984-10-11"), date("1985-11-25")),
duration.between(date("1984-10-11"), date("1984-10-11")),
duration.between(date("1984-10-11"), datetime("1984-10-12T1:40:32.142+0100")),
duration.between(date("2015-06-24"), localtime("14:30")),
duration.between(localdatetime("2015-07-21T1:40:32.142"), localdatetime("2016-07-21T1:45:22.142")),
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 424. Result**

<table>
<thead>
<tr>
<th>aDuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1Y1M14D</td>
</tr>
<tr>
<td>P-1Y-1M-14D</td>
</tr>
<tr>
<td>P1DT21H40M32.142S</td>
</tr>
<tr>
<td>PT14H30M</td>
</tr>
<tr>
<td>PT2H</td>
</tr>
<tr>
<td>PTY4M50S</td>
</tr>
<tr>
<td>PT1H</td>
</tr>
</tbody>
</table>

Rows: 7

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(instant_1, instant_2)`

**Returns:**

A `Duration`.

**Arguments:**
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>instant₁</td>
<td>An expression returning any temporal instant type (Date etc) that represents the starting instant.</td>
</tr>
<tr>
<td>instant₂</td>
<td>An expression returning any temporal instant type (Date etc) that represents the ending instant.</td>
</tr>
</tbody>
</table>

**Considerations:**

- **If** `instant₁` **occurs earlier than** `instant₂`, **the resulting Duration will be negative.**
- **If** `instant₁` **has a time component and** `instant₂` **does not, the time component of** `instant₁` **is assumed to be midnight, and vice versa.**
- **If** `instant₁` **has a time zone component and** `instant₂` **does not, the time zone component of** `instant₁` **is assumed to be the same as that of** `instant₂`, **and vice versa.**
- **If** `instant₁` **has a date component and** `instant₂` **does not, the date component of** `instant₁` **is assumed to be the same as that of** `instant₂`, **and vice versa.**
- **Any difference smaller than a whole month is disregarded.**

**Query**

```plaintext
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(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 425. Result**

<table>
<thead>
<tr>
<th>aDuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1Y1M</td>
</tr>
<tr>
<td>P-1Y-1M</td>
</tr>
<tr>
<td>PT0S</td>
</tr>
<tr>
<td>PT0S</td>
</tr>
<tr>
<td>P1Y</td>
</tr>
<tr>
<td>PT0S</td>
</tr>
</tbody>
</table>

**Rows: 6**

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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>instant</td>
<td>An expression returning any temporal instant type (Date etc) that represents the starting instant.</td>
</tr>
<tr>
<td>instant</td>
<td>An expression returning any temporal instant type (Date etc) that represents the ending instant.</td>
</tr>
</tbody>
</table>

Considerations:

If `instant` occurs earlier than `instant`, the resulting Duration will be negative.

If `instant` has a time component and `instant` does not, the time component of `instant` is assumed to be midnight, and vice versa.

If `instant` has a time zone component and `instant` does not, the time zone component of `instant` is assumed to be the same as that of `instant`, and vice versa.

If `instant` has a date component and `instant` does not, the date component of `instant` is assumed to be the same as that of `instant`, 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(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 426. Result

<table>
<thead>
<tr>
<th>aDuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>P410D</td>
</tr>
<tr>
<td>P-4100</td>
</tr>
<tr>
<td>P1D</td>
</tr>
<tr>
<td>PT0S</td>
</tr>
<tr>
<td>P366D</td>
</tr>
<tr>
<td>PT0S</td>
</tr>
</tbody>
</table>

Rows: 6
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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>instant1</td>
<td>An expression returning any temporal instant type (Date etc) that represents the starting instant.</td>
</tr>
<tr>
<td>instant2</td>
<td>An expression returning any temporal instant type (Date etc) that represents the ending instant.</td>
</tr>
</tbody>
</table>

Considerations:

If instant1 occurs earlier than instant2, the resulting Duration will be negative.

If instant1 has a time component and instant2 does not, the time component of instant1 is assumed to be midnight, and vice versa.

If instant1 has a time zone component and instant2 does not, the time zone component of instant1 is assumed to be the same as that of instant2, and vice versa.

If instant1 has a date component and instant2 does not, the date component of instant1 is assumed to be the same as that of instant2, and vice versa.

Query

```sql
UNWIND [
  duration.inSeconds(date("1984-10-11"), date("1984-10-12")),
  duration.inSeconds(date("1984-10-11"), 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 427. Result

<table>
<thead>
<tr>
<th>aDuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>PT24H</td>
</tr>
<tr>
<td>PT-24H</td>
</tr>
<tr>
<td>PT25H32.142S</td>
</tr>
<tr>
<td>PT14H30M</td>
</tr>
<tr>
<td>PT1H</td>
</tr>
</tbody>
</table>
10.11. 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:

- `point.distance()`
- `point.withinBBox()`
- `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.

Graph

10.11.1. point.distance()

`point.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:** `point.distance(point1, point2)`

**Returns:**
A Float.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>point1</td>
<td>A point in either a geographic or cartesian coordinate system.</td>
</tr>
<tr>
<td>point2</td>
<td>A point in the same CRS as point1.</td>
</tr>
</tbody>
</table>

**Considerations:**

- `point.distance(null, null), point.distance(null, point2) and point.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**

```sql
WITH point({ x: 2.3, y: 4.5, crs: 'cartesian'}) AS p1,
     point({ x: 1.1, y: 5.4, crs: 'cartesian'}) AS p2
RETURN point.distance(p1, p2) AS dist
```

The distance between two 2D points in the Cartesian CRS is returned.

**Table 428. Result**

<table>
<thead>
<tr>
<th>dist</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.5</td>
</tr>
</tbody>
</table>

**Query**

```sql
WITH point({ longitude: 12.78, latitude: 56.7, height: 100}) AS p1,
     point({ latitude: 56.71, longitude: 12.79, height: 100}) AS p2
RETURN point.distance(p1, p2) as dist
```

The distance between two 3D points in the WGS 84 CRS is returned.

**Table 429. Result**

<table>
<thead>
<tr>
<th>dist</th>
</tr>
</thead>
<tbody>
<tr>
<td>1269.9148706779097</td>
</tr>
</tbody>
</table>
The distance between the train station in Copenhagen and the Neo4j office in Malmo is returned.

Table 430. Result

<table>
<thead>
<tr>
<th>travelDistance</th>
<th>27842.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

If `null` is provided as one or both of the arguments, `null` is returned.

Table 431. Result

<table>
<thead>
<tr>
<th>d</th>
<th>&lt;null&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

10.11.2. point.withinBBox()

`point.withinBBox()` takes the following arguments:

- The point to check.
- The lower-left (south-west) point of a bounding box.
- The upper-right (or north-east) point of a bounding box.

The return value will be true if the provided point is contained in the bounding box (boundary included), otherwise the return value will be false.

**Syntax:** `point.withinBBox(point, lowerLeft, upperRight)`

**Returns:**

A Boolean.

**Arguments:**
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>point</td>
<td>A point in either a geographic or cartesian coordinate system.</td>
</tr>
<tr>
<td>lowerLeft</td>
<td>A point in the same CRS as 'point'.</td>
</tr>
<tr>
<td>upperRight</td>
<td>A point in the same CRS as 'point'.</td>
</tr>
</tbody>
</table>

**Considerations:**

- `point.withinBBox(p1, p2, p3)` will return `null` if any of the arguments evaluate to `null`.
- Attempting to use points with different Coordinate Reference Systems (such as WGS 84 2D and WGS 84 3D) will return `null`.
- `point.withinBBox` will handle crossing the 180th meridian in geographic coordinates.
- Switching the longitude of the `lowerLeft` and `upperRight` in geographic coordinates will switch the direction of the resulting bounding box.
- Switching the latitude of the `lowerLeft` and `upperRight` in geographic coordinates so that the former is north of the latter will result in an empty range.

**Query**

```sql
WITH point({x: 0, y: 0, crs: 'cartesian'}) AS lowerLeft,
     point({x: 10, y: 10, crs: 'cartesian'}) AS upperRight
RETURN point.withinBBox(point({x: 5, y: 5, crs: 'cartesian'}), lowerLeft, upperRight) AS result
```

Checking if a point in Cartesian CRS is contained in the bounding box.

### Table 432. Result

<table>
<thead>
<tr>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
</tr>
</tbody>
</table>

Rows: 1

**Query**

```sql
WITH point({longitude: 12.53, latitude: 55.66}) AS lowerLeft,
     point({longitude: 12.614, latitude: 55.70}) AS upperRight
MATCH (t:TrainStation)
WHERE point.withinBBox(point({longitude: t.longitude, latitude: t.latitude}), lowerLeft, upperRight)
RETURN count(t)
```

Finds all train stations contained in a bounding box around Copenhagen.

### Table 433. Result

<table>
<thead>
<tr>
<th>count(t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Rows: 1
Query

WITH point({longitude: 179, latitude: 55.66}) AS lowerLeft, point({longitude: -179, latitude: 55.70}) AS upperRight
RETURN point.withinBBox(point({longitude: 180, latitude: 55.66}), lowerLeft, upperRight) AS result

A bounding box that crosses the 180th meridian.

Table 434. Result

<table>
<thead>
<tr>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
</tr>
</tbody>
</table>

Rows: 1

Query

RETURN point.withinBBox(null, point({longitude: 56.7, latitude: 12.78}), point({longitude: 57.0, latitude: 13.0})) AS in

If null is provided as any of the arguments, null is returned.

Table 435. Result

<table>
<thead>
<tr>
<th>in</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

Rows: 1

10.11.3. 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:

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
</tr>
<tr>
<td>longitude/x</td>
</tr>
<tr>
<td>A numeric expression that represents the longitude/x value in decimal degrees</td>
</tr>
<tr>
<td>latitude/y</td>
</tr>
<tr>
<td>A numeric expression that represents the latitude/y value in decimal degrees</td>
</tr>
<tr>
<td>crs</td>
</tr>
<tr>
<td>The optional string 'WGS-84'</td>
</tr>
<tr>
<td>srid</td>
</tr>
<tr>
<td>The optional number 4326</td>
</tr>
</tbody>
</table>
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 436. Result**

<table>
<thead>
<tr>
<th>point</th>
</tr>
</thead>
<tbody>
<tr>
<td>point({x: 56.7, y: 12.78, crs: 'wgs-84'})</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

**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 437. Result**

<table>
<thead>
<tr>
<th>point</th>
</tr>
</thead>
<tbody>
<tr>
<td>point({x: 2.3, y: 4.5, crs: 'wgs-84'})</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

**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 438. Result**

<table>
<thead>
<tr>
<th>officePoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>point({x: 12.994341, y: 55.611784, crs: 'wgs-84'})</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

**Query**

```
RETURN point(null) AS p
```
If null is provided as the argument, null is returned.

Table 439. Result

<table>
<thead>
<tr>
<th>p</th>
<th>&lt;null&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows:</td>
<td>1</td>
</tr>
</tbody>
</table>

10.11.4. 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>longitude/x</td>
<td>A numeric expression that represents the longitude/x value in decimal degrees</td>
</tr>
<tr>
<td>latitude/y</td>
<td>A numeric expression that represents the latitude/y value in decimal degrees</td>
</tr>
<tr>
<td>height/z</td>
<td>A numeric expression that represents the height/z value in meters</td>
</tr>
<tr>
<td>crs</td>
<td>The optional string 'WGS-84-3D'</td>
</tr>
<tr>
<td>srid</td>
<td>The optional number 4979</td>
</tr>
</tbody>
</table>

Considerations:

If any argument provided to point() is null, null will be returned.

If the height/z key and value is not provided, a 2D point in the WGS 84 CRS will be returned.

If the coordinates are specified using latitude and longitude, the crs or srid fields are optional and inferred to be 'WGS-84-3D' (srid=4979).

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, 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
Table 440. Result

<table>
<thead>
<tr>
<th>point</th>
</tr>
</thead>
<tbody>
<tr>
<td>point({x: 56.7, y: 12.78, z: 8.0, crs: 'wgs-84-3d'})</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

10.11.5. point() - Cartesian 2D

point({x, y [, crs][, srid]}) 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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>A numeric expression</td>
</tr>
<tr>
<td>y</td>
<td>A numeric expression</td>
</tr>
<tr>
<td>crs</td>
<td>The optional string 'cartesian'</td>
</tr>
<tr>
<td>srid</td>
<td>The optional number 7203</td>
</tr>
</tbody>
</table>

**Considerations:**

If any argument provided to point() is null, null will be returned.

The crs or srid fields are optional and default to the Cartesian CRS (which means srid:7203).

**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 441. Result

<table>
<thead>
<tr>
<th>point</th>
</tr>
</thead>
<tbody>
<tr>
<td>point({x: 2.3, y: 4.5, crs: 'cartesian'})</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>
10.11.6. point() - Cartesian 3D

point({x, y, z, [, crs][, srid]}) 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>A numeric expression</td>
</tr>
<tr>
<td>y</td>
<td>A numeric expression</td>
</tr>
<tr>
<td>z</td>
<td>A numeric expression</td>
</tr>
<tr>
<td>crs</td>
<td>The optional string 'cartesian-3D'</td>
</tr>
<tr>
<td>srid</td>
<td>The optional number 9157</td>
</tr>
</tbody>
</table>

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 442. Result

<table>
<thead>
<tr>
<th>point</th>
</tr>
</thead>
<tbody>
<tr>
<td>point({x: 2.3, y: 4.5, z: 2.0, crs: 'cartesian-3d'})</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

10.12. LOAD CSV functions

LOAD CSV functions can be used to get information about the file that is processed by LOAD CSV.
The functions described on this page are only useful when run on a query that uses `LOAD CSV`. In all other contexts they will always return `null`.

Functions:
- `linenumber()`
- `file()`

10.12.1. `linenumber()`

`linenumber()` returns the line number that `LOAD CSV` is currently using.

Syntax: `linenumber()`

Returns:
- An Integer.

Considerations:
- `null` will be returned if this function is called without a `LOAD CSV` context.
- If the CSV file contains headers, the headers will be `linenumber` 1 and the 1st row of data will have a `linenumber` of 2.

10.12.2. `file()`

`file()` returns the absolute path of the file that `LOAD CSV` is using.

Syntax: `file()`

Returns:
- A String.

Considerations:
- `null` will be returned if this function is called without a `LOAD CSV` context.

10.13. 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.

There are two main types of functions that can be developed and used:
### 10.13.1. User-defined scalar functions

For each incoming row the function takes parameters and returns a single result.

This example shows how you invoke a user-defined function called `join` from Cypher.

**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 443. Result**

<table>
<thead>
<tr>
<th>members</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>John,Paul,George,Ringo</em></td>
</tr>
</tbody>
</table>

Rows: 1

For developing and deploying user-defined functions in Neo4j, see [*Extending Neo4j → User-defined functions*](#).
Chapter 11. User-defined aggregation functions

Aggregating functions consume many rows and produces a single aggregated result.

This example shows how you invoke a user-defined aggregation function called `longestString` from Cypher.

11.1. Call a user-defined aggregation function

This calls the user-defined function `org.neo4j.function.example.longestString()`.

Query

```
MATCH (n:Member)
RETURN org.neo4j.function.example.longestString(n.name) AS member
```

Table 444. Result

<table>
<thead>
<tr>
<th>member</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;George&quot;</td>
</tr>
</tbody>
</table>

Rows: 1
Chapter 12. Indexes for search performance

This section explains how to manage indexes used for search performance.

This section describes how to manage indexes. For query performance purposes, it is important to also understand how the indexes are used by the Cypher planner. Refer to Query tuning for examples and in-depth discussions on how query plans result from different index and query scenarios. See specifically The use of indexes for examples of how various index scenarios result in different query plans.

For information on index configuration and limitations, refer to Operations Manual → Index configuration.

12.1. Indexes (types and limitations)

A database index is a redundant copy of some of the data in the database for the purpose of making searches of related 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.

Once an index has been created, it will be managed and kept up to date by the DBMS. Neo4j will automatically pick up and start using the index once it has been created and brought online.

There are multiple index types available: b-tree (deprecated), fulltext, lookup, and text index types. See Full-text search index for more information about fulltext indexes. Token lookup indexes contain nodes with one or more labels or relationship types, without regard for any properties.

Cypher enables the creation of b-tree indexes on one or more properties for all nodes or relationships with a given label or relationship type:

- An index created on a single property for any given label or relationship type is called a single-property index.
- An index created on more than one property for any given label or relationship type is called a composite index.

Differences in the usage patterns between composite and single-property indexes are described in Composite index limitations.

Additionally, a text index is a kind of single-property index, with the limitation that it only recognizes properties with string values. Nodes or relationships with the indexed label or relationship type where the indexed property is of another value type are not included in the index.

The following is true for indexes:

- Best practice is to give the index a name when it is created. If the index is not explicitly named, it gets an auto-generated name.
- The index name must be unique among both indexes and constraints.
- Index creation is by default not idempotent, and an error will be thrown if you attempt to create the same index twice. Using the keyword IF NOT EXISTS makes the command idempotent, and no error will be thrown if you attempt to create the same index twice.
12.1.1. Syntax

Table 445. Syntax for managing indexes
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE [BTREE] INDEX [index_name] [IF NOT EXISTS] FOR (n:LabelName) ON (n.propertyName) [OPTIONS &quot;{&quot; option: value[,...] &quot;}&quot;]</td>
<td>Create a single-property index on nodes. Index provider and configuration can be specified using the OPTIONS clause.</td>
<td>Explicit use of BTREE keyword or b-tree options are deprecated in 4.4 and will be replaced in 5.0. Best practice is to give the index a name when it is created. If the index is not explicitly named, it gets an auto-generated name. The index name must be unique among both indexes and constraints. The command is optionally idempotent, with the default behavior to throw an error if you attempt to create the same index twice. With IF NOT EXISTS, no error is thrown and nothing happens should an index with the same name or same schema and index type already exist. It may still throw an error if conflicting constraints exist, such as constraints with the same name or schema and backing index type.</td>
</tr>
<tr>
<td>CREATE [BTREE] INDEX [index_name] [IF NOT EXISTS] FOR ()-&quot;[r:TYPE_NAME&quot;]&quot;-() ON (r.propertyName) [OPTIONS &quot;{&quot; option: value[,...] &quot;}&quot;]</td>
<td>Create a single-property index on relationships. Index provider and configuration can be specified using the OPTIONS clause. Explicit use of BTREE keyword or b-tree options are deprecated in 4.4 and will be replaced in 5.0.</td>
<td></td>
</tr>
<tr>
<td>CREATE [BTREE] INDEX [index_name] [IF NOT EXISTS] FOR (n:LabelName) ON (n.propertyName_1, n.propertyName_2, n.propertyName_n) [OPTIONS &quot;{&quot; option: value[,...] &quot;}&quot;]</td>
<td>Create a composite index on nodes. Index provider and configuration can be specified using the OPTIONS clause. Explicit use of BTREE keyword or b-tree options are deprecated in 4.4 and will be replaced in 5.0.</td>
<td></td>
</tr>
<tr>
<td>CREATE [BTREE] INDEX [index_name] [IF NOT EXISTS] FOR ()-&quot;[r:TYPE_NAME&quot;]&quot;-() ON (r.propertyName_1, r.propertyName_2, r.propertyName_n) [OPTIONS &quot;{&quot; option: value[,...] &quot;}&quot;]</td>
<td>Create a composite index on relationships. Index provider and configuration can be specified using the OPTIONS clause. Explicit use of BTREE keyword or b-tree options are deprecated in 4.4 and will be replaced in 5.0.</td>
<td></td>
</tr>
<tr>
<td>CREATE LOOKUP INDEX [index_name] [IF NOT EXISTS] FOR (n) ON EACH labels(n) [OPTIONS &quot;{&quot; option: value[,...] &quot;}&quot;]</td>
<td>Create a node label lookup index. Index provider can be specified using the OPTIONS clause.</td>
<td></td>
</tr>
<tr>
<td>CREATE LOOKUP INDEX [index_name] [IF NOT EXISTS] FOR ()-&quot;[r&quot;]&quot;-() ON EACH type(r) [OPTIONS &quot;{&quot; option: value[,...] &quot;}&quot;]</td>
<td>Create a relationship type lookup index. Index provider can be specified using the OPTIONS clause.</td>
<td></td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
<td>Comment</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>CREATE TEXT INDEX [index_name] [IF NOT EXISTS] FOR (n:LabelName) ON (n.propertyName) [OPTIONS &quot;{option: value, ...}&quot;&quot;]</td>
<td>Create a text index on nodes where the property has a string value. Index provider can be specified using the OPTIONS clause.</td>
<td></td>
</tr>
<tr>
<td>CREATE TEXT INDEX [index_name] [IF NOT EXISTS] FOR ()-&quot;[r:TYPE_NAME&quot;]&quot;-() ON (r.propertyName) [OPTIONS &quot;{option: value, ...}&quot;&quot;]</td>
<td>Create a text index on relationships where the property has a string value. Index provider can be specified using the OPTIONS clause.</td>
<td></td>
</tr>
<tr>
<td>DROP INDEX index_name [IF EXISTS]</td>
<td>Drop an index of any index type. The command is optionally idempotent, with the default behavior to throw an error if you attempt to drop the same index twice. With IF EXISTS, no error is thrown and nothing happens should the index not exist.</td>
<td></td>
</tr>
<tr>
<td>SHOW [ALL</td>
<td>Btree</td>
<td>Fulltext</td>
</tr>
<tr>
<td>DROP INDEX ON :LabelName(propertyName)</td>
<td>Drop a single-property index on nodes without specifying a name. This syntax is deprecated.</td>
<td></td>
</tr>
<tr>
<td>DROP INDEX ON :LabelName (n.propertyName_1, n.propertyName_2, n.propertyName_n)</td>
<td>Drop a composite index on nodes without specifying a name.</td>
<td></td>
</tr>
</tbody>
</table>

Creating an index requires the CREATE INDEX privilege, while dropping an index requires the DROP INDEX privilege and listing indexes require the SHOW INDEX privilege.

Planner hints and the USING keyword describes how to make the Cypher planner use specific indexes (especially in cases where the planner would not necessarily have used them).

### 12.1.2. Composite index limitations

Like single-property b-tree indexes, composite b-tree indexes support all predicates:
• equality check: \( n.prop = value \)
• list membership check: \( n.prop \text{ IN list} \)
• existence check: \( n.prop \text{ IS NOT NULL} \)
• range search: \( n.prop \text{ > value} \)
• prefix search: \( \text{STARTS WITH} \)
• suffix search: \( \text{ENDS WITH} \)
• substring search: \( \text{CONTAINS} \)

For details about each operator, see Operators.

However, predicates might be planned as existence check and a filter. For most predicates, this can be avoided by following these restrictions:

• If there is any equality check and list membership check predicates, they need to be for the first properties defined by the index.
• There can be up to one range search or prefix search predicate.
• There can be any number of existence check predicates.
• Any predicate after a range search, prefix search or existence check predicate has to be an existence check predicate.

However, the suffix search and substring search predicates are always planned as existence check and a filter and any predicates following after will therefore also be planned as such.

For example, an index on nodes with \( :\text{Label}(\text{prop1,prop2,prop3,prop4,prop5,prop6}) \) and predicates:

```
WHERE n.prop1 = 'x' AND n.prop2 = 1 AND n.prop3 > 5 AND n.prop4 < 'e' AND n.prop5 = true AND n.prop6 IS NOT NULL
```

will be planned as:

```
WHERE n.prop1 = 'x' AND n.prop2 = 1 AND n.prop3 > 5 AND n.prop4 IS NOT NULL AND n.prop5 IS NOT NULL AND n.prop6 IS NOT NULL
```

with filters on \( n.prop4 < 'e' \) and \( n.prop5 = true \), since \( n.prop3 \) has a range search predicate.

And an index on nodes with \( :\text{Label}(\text{prop1,prop2}) \) with predicates:

```
WHERE n.prop1 ENDS WITH 'x' AND n.prop2 = false
```

will be planned as:

```
WHERE n.prop1 IS NOT NULL AND n.prop2 IS NOT NULL
```

with filters on \( n.prop1 \text{ ENDS WITH 'x'} \) and \( n.prop2 = false \), since \( n.prop1 \) has a suffix search predicate.
Composite indexes require predicates on all properties indexed. If there are predicates on only a subset of the indexed properties, it will not be possible to use the composite index. To get this kind of fallback behavior, it is necessary to create additional indexes on the relevant sub-set of properties or on single properties.

### 12.2. Creating indexes

#### 12.2.1. Create a single-property b-tree index for nodes

A named b-tree index on a single property for all nodes with a particular label can be created with `CREATE INDEX index_name FOR (n:Label) ON (n.property)`. Note that the index is not immediately available, but is created in the background.

**Query**

```sql
CREATE INDEX node_index_name FOR (n:Person) ON (n.surname)
```

Note that the index name must be unique.

**Result**

```
+-------------------+
| No data returned. |
| Indexes added: 1  |
+-------------------+
```

#### 12.3. Create a single-property b-tree index for relationships

A named b-tree index on a single property for all relationships with a particular relationship type can be created with `CREATE INDEX index_name FOR ()-[r:TYPE]-() ON (r.property)`. Note that the index is not immediately available, but is created in the background.

**Query**

```sql
CREATE INDEX rel_index_name FOR ()-[r:KNOWS]-() ON (r.since)
```

Note that the index name must be unique.

**Result**

```
+-------------------+
| No data returned. |
| Indexes added: 1  |
+-------------------+
```

#### 12.4. Create a single-property b-tree index only if it does not already exist

If it is not known whether an index exists or not, add `IF NOT EXISTS` to ensure it does.
Note that the index will not be created if there already exists an index with the same schema and type, same name or both.

12.5. Create a single-property b-tree index with specified index provider

To create a single property b-tree index with a specific index provider, the OPTIONS clause is used. Valid values for the index provider are native-btree-1.0 and lucene+native-3.0, default is native-btree-1.0.

Query

```
CREATE BTREE INDEX index_with_provider FOR ()-[r:TYPE]-() ON (r.prop1) OPTIONS {indexProvider: 'native-btree-1.0'}
```

Can be combined with specifying index configuration.

Result

```
<table>
<thead>
<tr>
<th>No data returned.</th>
</tr>
</thead>
</table>
Indexes added: 1
```

12.6. Create a single-property b-tree index with specified index configuration

To create a single property b-tree index with a specific index configuration, the OPTIONS clause is used. The valid configuration settings are

- `spatial.cartesian.min`
- `spatial.cartesian.max`
- `spatial.cartesian-3d.min`
- `spatial.cartesian-3d.max`
- `spatial.wgs-84.min`
- `spatial.wgs-84.max`
- `spatial.wgs-84-3d.min`
- `spatial.wgs-84-3d.max`
Non-specified settings have their respective default values.

Query

```sql
CREATE BTREE INDEX index_with_config FOR (n:Label) ON (n.prop2)
OPTIONS (indexConfig: {'spatial.cartesian.min': [-100.0, -100.0], 'spatial.cartesian.max': [100.0, 100.0]})
```

Can be combined with specifying index provider.

Result

| No data returned. |
| Indexes added: 1 |

12.7. Create a composite b-tree index for nodes

A named b-tree index on multiple properties for all nodes with a particular label — i.e. a composite index — can be created with `CREATE INDEX index_name FOR (n:Label) ON (n.prop1, ..., n.propN)`. Only nodes with the specified label and that contain all the properties in the index definition will be added to the index. Note that the composite index is not immediately available, but is created in the background. The following statement will create a named composite index on all nodes labeled with `Person` and which have both an `age` and `country` property:

Query

```sql
CREATE INDEX node_index_name FOR (n:Person) ON (n.age, n.country)
```

Note that the index name must be unique.

Result

| No data returned. |
| Indexes added: 1 |

12.8. Create a composite b-tree index for relationships

A named b-tree index on multiple properties for all relationships with a particular relationship type — i.e. a composite index — can be created with `CREATE INDEX index_name FOR ()-[r:TYPE]-() ON (r.prop1, ..., r.propN)`. Only relationships with the specified type and that contain all the properties in the index definition will be added to the index. Note that the composite index is not immediately available, but is created in the background. The following statement will create a named composite index on all relationships labeled with `PURCHASED` and which have both a `date` and `amount` property:

Query

```sql
CREATE INDEX rel_index_name FOR ()-[r:PURCHASED]-() ON (r.date, r.amount)
```
12.9. Create a composite b-tree index with specified index provider and configuration

To create a composite b-tree index with a specific index provider and configuration, the **OPTIONS** clause is used. Valid values for the index provider are **native-btree-1.0** and **lucene+native-3.0**, default is **native-btree-1.0**. The valid configuration settings are

- `spatial.cartesian.min`
- `spatial.cartesian.max`
- `spatial.cartesian-3d.min`
- `spatial.cartesian-3d.max`
- `spatial.wgs-84.min`
- `spatial.wgs-84.max`
- `spatial.wgs-84-3d.min`
- `spatial.wgs-84-3d.max`

Non-specified settings have their respective default values.

**Query**

```
CREATE INDEX index_with_options
FOR (n:Label) ON (n.prop1, n.prop2)
OPTIONS {
  indexProvider: 'lucene+native-3.0',
  indexConfig: {
    'spatial.wgs-84.min': [-100.0, -80.0],
    'spatial.wgs-84.max': [100.0, 80.0]
  }
}
```

Specifying index provider and configuration can be done individually.

**Result**

<table>
<thead>
<tr>
<th>No data returned.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexes added: 1</td>
</tr>
</tbody>
</table>

12.10. Create a node label lookup index

A named token lookup index for all nodes with one or more labels can be created with **CREATE LOOKUP INDEX index_name FOR (n) ON EACH labels(n)**. Note that the index is not immediately available, but is created in the background.
12.11. Create a relationship type lookup index

A named token lookup index for all relationships with any relationship type can be created with `CREATE LOOKUP INDEX index_name FOR ()-[r]-() ON EACH type(r)`. Note that the index is not immediately available, but is created in the background.

Query

```
CREATE LOOKUP INDEX rel_type_lookup_index FOR ()-[r]-() ON EACH type(r)
```

Note that it can only be created once and that the index name must be unique.

Result

```
<table>
<thead>
<tr>
<th>No data returned.</th>
</tr>
</thead>
</table>
Indexes added: 1
```

12.12. Create a token lookup index specifying the index provider

Token lookup indexes allow setting the index provider using the `OPTIONS` clause. Only one valid value exists for the index provider, `token-lookup-1.0`, which is the default value.

Query

```
CREATE LOOKUP INDEX node_label_lookup_index_2 FOR (n) ON EACH labels(n) OPTIONS {
  'indexProvider: token-lookup-1.0'
```

There is no supported index configuration for token lookup indexes.

Result

```
<table>
<thead>
<tr>
<th>No data returned.</th>
</tr>
</thead>
</table>
Indexes added: 1
```
12.13. Create a node text index

A named text index on a single property for all nodes with a particular label can be created with `CREATE TEXT INDEX index_name FOR (n:Label) ON (n.property)`. Note that the index is not immediately available, but is created in the background.

Query

```
CREATE TEXT INDEX node_index_name FOR (n:Person) ON (n.nickname)
```

Note that text indexes only recognize string values, do not support multiple properties, and that the index name must be unique.

Result

```
+-------------------+
| No data returned. |
+-------------------+
Indexes added: 1
```

12.14. Create a relationship text index

A named text index on a single property for all relationships with a particular relationship type can be created with `CREATE TEXT INDEX index_name FOR ()-[r:TYPE]-() ON (r.property)`. Note that the index is not immediately available, but is created in the background.

Query

```
CREATE TEXT INDEX rel_index_name FOR ()-[r:KNOWS]-() ON (r.interest)
```

Note that text indexes only recognize string values, do not support multiple properties, and that the index name must be unique.

Result

```
+-------------------+
| No data returned. |
+-------------------+
Indexes added: 1
```

12.15. Create a text index only if it does not already exist

If it is not known whether an index exists or not, add `IF NOT EXISTS` to ensure it does.

Query

```
CREATE TEXT INDEX node_index_name IF NOT EXISTS FOR (n:Person) ON (n.nickname)
```

Note that the index will not be created if there already exists an index with the same schema and type, same name or both.
12.16. Create a text index specifying the index provider

To create a text index with a specific index provider, the OPTIONS clause is used. Only one valid value exists for the index provider, **text-1.0**, which is the default value.

**Query**

```plaintext
CREATE TEXT INDEX index_with_provider FOR ()-[r:TYPE]-() ON (r.prop1) OPTIONS {
    indexProvider: 'text-1.0'
}
```

There is no supported index configuration for text indexes.

**Result**

```
<table>
<thead>
<tr>
<th>No data returned.</th>
</tr>
</thead>
</table>
Indexes added: 1
```

12.17. Failure to create an already existing index

Create an index on the property **title** on nodes with the **Book** label, when that index already exists.

**Query**

```plaintext
CREATE INDEX bookTitleIndex FOR (book:Book) ON (book.title)
```

In this case the index can’t be created because it already exists.

**Error message**

```
There already exists an index (:Book {title}).
```

12.18. Failure to create an index with the same name as an already existing index

Create a named index on the property **numberOfPages** on nodes with the **Book** label, when an index with that name already exists.

**Query**

```plaintext
CREATE INDEX indexOnBooks FOR (book:Book) ON (book.numberOfPages)
```

In this case the index can’t be created because there already exists an index with that name.
Error message

There already exists an index called 'indexOnBooks'.

12.19. Failure to create an index when a constraint already exists

Create an index on the property isbn on nodes with the Book label, when an index-backed constraint already exists on that schema.

Query

```
CREATE INDEX bookIsbnIndex FOR (book:Book) ON (book.isbn)
```

In this case the index can’t be created because a index-backed constraint already exists on that label and property combination.

Error message

There is a uniqueness constraint on (:Book {isbn}), so an index is already created that matches this.

12.20. Failure to create an index with the same name as an already existing constraint

Create a named index on the property numberOfPages on nodes with the Book label, when a constraint with that name already exists.

Query

```
```

In this case the index can’t be created because there already exists a constraint with that name.

Error message

There already exists a constraint called 'bookRecommendations'.
Chapter 13. Listing indexes

Listing indexes can be done with `SHOW INDEXES`, which will produce a table with the following columns:

Table 446. List indexes output

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Default output</th>
<th>Full output</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The id of the index.</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>name</td>
<td>Name of the index (explicitly set by the user or automatically assigned).</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>state</td>
<td>Current state of the index.</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>populationPercent</td>
<td>% of index population.</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>uniqueness</td>
<td>Tells if the index is only meant to allow one value per key.</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>type</td>
<td>The IndexType of this index ([STREE, FULLTEXT, LOOKUP, or TEXT]).</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>entityType</td>
<td>Type of entities this index represents (nodes or relationship).</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>labelsOrTypes</td>
<td>The labels or relationship types of this index.</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>properties</td>
<td>The properties of this index.</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>indexProvider</td>
<td>The index provider for this index.</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>options</td>
<td>The options passed to CREATE command.</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>failureMessage</td>
<td>The failure description of a failed index.</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>createStatement</td>
<td>Statement used to create the index.</td>
<td>+</td>
<td></td>
</tr>
</tbody>
</table>

Listing indexes also allows for `WHERE` and `YIELD` clauses to filter the returned rows and columns.

While the command for listing indexes require the `SHOW INDEX` privilege, the deprecated built-in procedures for listing indexes, such as `db.indexes`, work as before and are not affected by the privilege.
13.1. Listing indexes examples

13.1.1. Listing all indexes

To list all indexes with the default output columns, the `SHOW INDEXES` command can be used. If all columns are required, use `SHOW INDEXES YIELD *`.

Query

```sql
SHOW INDEXES
```

One of the output columns from `SHOW INDEXES` is the name of the index. This can be used to drop the index with the `DROP INDEX` command.

Result

```
+---------------------------------------------------------------------------------------------------------+----------------------------------------+
| id | name             | state    | populationPercent | uniqueness | type    | entityType |
|    | labelsOrTypes   | properties | indexProvider      |            |        |            |
+---------------------------------------------------------------------------------------------------------+----------------------------------------+
| 4  | "index_44d2128f" | "ONLINE" | 100.0             | "NONUNIQUE" | "BTREE" | "NODE"     |
|    | "middlename"    | "native-btree-1.0" |            |            |        |
| 7  | "index_58a1c03e" | "ONLINE" | 100.0             | "NONUNIQUE" | "BTREE" | "NODE"     |
|    | "location"      | "native-btree-1.0" |            |            |        |
| 5  | "index_763f72db" | "ONLINE" | 100.0             | "NONUNIQUE" | "TEXT"  | "NODE"     |
|    | "middlename"    | "text-1.0"  |            |            |        |
| 5  | "index_d7c12ba3" | "ONLINE" | 100.0             | "NONUNIQUE" | "BTREE" | "NODE"     |
|    | "highScore"     | "native-btree-1.0" |            |            |        |
| 3  | "index_deeafdb2" | "ONLINE" | 100.0             | "NONUNIQUE" | "BTREE" | "NODE"     |
|    | "firstname"     | "native-btree-1.0" |            |            |        |
| 6  | "index_eadb868e" | "ONLINE" | 100.0             | "NONUNIQUE" | "TEXT"  | "NODE"     |
|    | "surname"       | "text-1.0"  |            |            |        |
+---------------------------------------------------------------------------------------------------------+----------------------------------------+
```

6 rows

13.1.2. Listing indexes with filtering

One way of filtering the output from `SHOW INDEXES` by index type is the use of type keywords, listed in the syntax table. For example, to show only b-tree indexes, use `SHOW BTREE INDEXES`. Another more flexible way of filtering the output is to use the `WHERE` clause. An example is to only show indexes not belonging to constraints.

Query

```sql
SHOW BTREE INDEXES WHERE uniqueness = 'NONUNIQUE'
```

This will only return the default output columns. To get all columns, use `SHOW INDEXES YIELD * WHERE ...`.
<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>state</th>
<th>populationPercent</th>
<th>uniqueness</th>
<th>type</th>
<th>entityType</th>
</tr>
</thead>
<tbody>
<tr>
<td>labelsOrTypes</td>
<td>properties</td>
<td>indexProvider</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| +-------------------------------------------------------------------------------------------------------+

| 4   | "index_44d2128f" | "ONLINE" | 100.0             | "NONUNIQUE" | "BTREE" | "NODE"         |
| ["Person"] | ["middlename"] | "native-btree-1.0" |
| 7   | "index_58a1c03e" | "ONLINE" | 100.0             | "NONUNIQUE" | "BTREE" | "NODE"         |
| ["Person"] | ["location"] | "native-btree-1.0" |
| 9   | "index_c207e3e6" | "ONLINE" | 100.0             | "NONUNIQUE" | "BTREE" | "RELATIONSHIP" |
| ["KNOWS"] | ["since"] | "native-btree-1.0" |
| 8   | "index_d7c12ba3" | "ONLINE" | 100.0             | "NONUNIQUE" | "BTREE" | "NODE"         |
| ["Person"] | ["highScore"] | "native-btree-1.0" |
| 3   | "index_deeafdb2" | "ONLINE" | 100.0             | "NONUNIQUE" | "BTREE" | "NODE"         |
| ["Person"] | ["firstname"] | "native-btree-1.0" |

5 rows
Chapter 14. Deleting indexes

14.1. Drop an index

An index can be dropped using the name with the `DROP INDEX index_name` command. This command can drop indexes of any type, except those backing constraints. The name of the index can be found using the `SHOW INDEXES` command, given in the output column `name`.

**Query**

```
DROP INDEX index_name
```

**Result**

```
| No data returned. |
Indexes removed: 1
```

14.2. Drop a non-existing index

If it is uncertain if an index exists and you want to drop it if it does but not get an error should it not, use:

**Query**

```
DROP INDEX missing_index_name IF EXISTS
```

**Result**

```
| No data returned, and nothing was changed. |
```
Chapter 15. Future indexes

Two new types of indexes, point and range indexes, will be introduced in a future release. They cannot be used in queries yet, but they can be created and dropped for migration purposes. These new index types together with text indexes will replace the current b-tree indexes. For more details on these new types, see the Operations Manual → Future indexes.

Like b-tree indexes, range indexes are created on one or more properties for all nodes or relationships with a given label or relationship type:

- An index created on a single property for any given label or relationship type is called a single-property index.
- An index created on more than one property for any given label or relationship type is called a composite index.

The differences in the usage patterns between composite and single-property indexes described in Composite index limitations also applies to range indexes.

Similar to b-tree indexes, range indexes may also back constraints by giving the range index provider when creating an index-backed constraint.

Point indexes, like text indexes, are a kind of single-property indexes, with the limitation that they only recognize properties with point values. Nodes or relationships with the indexed label or relationship type where the indexed property is of another value type are not included in the index.

15.1. Syntax

Table 447. Syntax for managing future indexes
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE RANGE INDEX [index_name] [IF NOT EXISTS] FOR (n:LabelName) ON (n.propertyName_1, n.propertyName_2, ...) [OPTIONS &quot;{&quot;option: value, ...}&quot;]</td>
<td>Create a range index on nodes, either on a single property or composite. Index provider can be specified using the OPTIONS clause.</td>
<td>Best practice is to give the index a name when it is created. If the index is not explicitly named, it gets an auto-generated name. The index name must be unique among both indexes and constraints. The command is optionally idempotent, with the default behavior to throw an error if you attempt to create the same index twice. With IF NOT EXISTS, no error is thrown and nothing happens should an index with the same name or same schema and index type already exist. It may still throw an error if conflicting constraints exist, such as constraints with the same name or schema and backing index type.</td>
</tr>
<tr>
<td>CREATE RANGE INDEX [index_name] [IF NOT EXISTS] FOR ()-[&quot;r:TYPE_NAME&quot;]-() ON (r.propertyName_1, r.propertyName_2, ...) [OPTIONS &quot;{&quot;option: value, ...}&quot;]</td>
<td>Create a range index on relationships, either on a single property or composite. Index provider can be specified using the OPTIONS clause.</td>
<td></td>
</tr>
<tr>
<td>CREATE POINT INDEX [index_name] [IF NOT EXISTS] FOR (n:LabelName) ON (n.propertyName) [OPTIONS &quot;{&quot;option: value, ...}&quot;]</td>
<td>Create a point index on nodes where the property has a point value. Index provider and configuration can be specified using the OPTIONS clause.</td>
<td></td>
</tr>
<tr>
<td>CREATE POINT INDEX [index_name] [IF NOT EXISTS] FOR ()-[&quot;r:TYPE_NAME&quot;]-() ON (r.propertyName) [OPTIONS &quot;{&quot;option: value, ...}&quot;]</td>
<td>Create a point index on relationships where the property has a point value. Index provider and configuration can be specified using the OPTIONS clause.</td>
<td></td>
</tr>
<tr>
<td>DROP INDEX index_name [IF EXISTS]</td>
<td>Drop an index of any index type. This is the same command as for the existing indexes.</td>
<td>The command is optionally idempotent, with the default behavior to throw an error if you attempt to drop the same index twice. With IF EXISTS, no error is thrown and nothing happens should the index not exist.</td>
</tr>
<tr>
<td>SHOW [ALL]BTREE</td>
<td>FULLTEXT</td>
<td>LOOKUP</td>
</tr>
</tbody>
</table>
15.2. Create index examples

15.3. Create a single-property range index for nodes

A named range index on a single property for all nodes with a particular label can be created with `CREATE RANGE INDEX index_name FOR (n:Label) ON (n.property)`. Note that the index is not immediately available, but is created in the background.

Query

```
CREATE RANGE INDEX node_range_index_name FOR (n:Person) ON (n.surname)
```

Note that the index name must be unique.

Result

```
+-------------------+
| No data returned. |
|                   |
| Indexes added: 1  |
+-------------------+
```

15.4. Create a single-property range index for relationships

A named range index on a single property for all relationships with a particular relationship type can be created with `CREATE RANGE INDEX index_name FOR ()-[r:TYPE]-() ON (r.property)`. Note that the index is not immediately available, but is created in the background.

Query

```
CREATE RANGE INDEX rel_range_index_name FOR ()-[r:KNOWS]-() ON (r.since)
```

Note that the index name must be unique.

Result

```
+-------------------+
| No data returned. |
|                   |
| Indexes added: 1  |
+-------------------+
```

15.5. Create a range index only if it does not already exist

If it is not known whether an index exists or not, add `IF NOT EXISTS` to ensure it does.

Query

```
CREATE RANGE INDEX node_range_index_name IF NOT EXISTS FOR (n:Person) ON (n.surname)
```

Note that the index will not be created if there already exists an index with the same schema and type, same name or both.
15.6. Create a range index specifying the index provider

To create a range index with a specific index provider, the `OPTIONS` clause is used. Only one valid value exists for the index provider, `range-1.0`, which is the default value.

Query

```
CREATE RANGE INDEX range_index_with_provider FOR ()-[r:TYPE]-() ON (r.prop1) OPTIONS {indexProvider: 'range-1.0'}
```

There is no supported index configuration for range indexes.

Result

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

Indexes added: 1
```

15.7. Create a composite range index for nodes

A named range index on multiple properties for all nodes with a particular label — i.e. a composite index — can be created with `CREATE RANGE INDEX index_name FOR (n:Label) ON (n.prop1, ..., n.propN)`. Only nodes with the specified label and that contain all the properties in the index definition will be added to the index. Note that the composite index is not immediately available, but is created in the background. The following statement will create a named composite range index on all nodes labeled with `Person` and which have both an `age` and `country` property:

Query

```
CREATE RANGE INDEX composite_range_node_index_name FOR (n:Person) ON (n.age, n.country)
```

Note that the index name must be unique.

Result

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

Indexes added: 1
```

15.8. Create a composite range index for relationships

A named range index on multiple properties for all relationships with a particular relationship type — i.e. a composite index — can be created with `CREATE RANGE INDEX index_name FOR ()-[r:TYPE]-() ON (r.prop1, ..., r.propN)`. Only relationships with the specified type and that contain all the properties in the
index definition will be added to the index. Note that the composite index is not immediately available, but is created in the background. The following statement will create a named composite range index on all relationships labeled with PURCHASED and which have both a date and amount property:

Query

```
CREATE RANGE INDEX composite_range_rel_index_name FOR ()-[r:PURCHASED]-() ON (r.date, r.amount)
```

Note that the index name must be unique.

Result

```
+-------------------+
| No data returned. |
| Indexes added: 1  |
+-------------------+
```

15.9. Create a node point index

A named point index on a single property for all nodes with a particular label can be created with CREATE POINT INDEX index_name FOR (n:Label) ON (n.property). Note that the index is not immediately available, but is created in the background.

Query

```
CREATE POINT INDEX node_index_name FOR (n:Person) ON (n.location)
```

Note that point indexes only recognize point values, do not support multiple properties, and that the index name must be unique.

Result

```
+-------------------+
| No data returned. |
| Indexes added: 1  |
+-------------------+
```

15.10. Create a relationship point index

A named point index on a single property for all relationships with a particular relationship type can be created with CREATE POINT INDEX index_name FOR ()-[r:TYPE]-() ON (r.property). Note that the index is not immediately available, but is created in the background.

Query

```
CREATE POINT INDEX rel_index_name FOR ()-[r:STREET]-() ON (r.intersection)
```

Note that point indexes only recognize point values, do not support multiple properties, and that the index name must be unique.
15.11. Create a point index only if it does not already exist

If it is not known whether an index exists or not, add **IF NOT EXISTS** to ensure it does.

Query

```
CREATE POINT INDEX node_index_name IF NOT EXISTS FOR (n:Person) ON (n.location)
```

Note that the index will not be created if there already exists an index with the same schema and type, same name or both.

Result

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

Indexes added: 1

15.12. Create a point index specifying the index provider

To create a point index with a specific index provider, the **OPTIONS** clause is used. Only one valid value exists for the index provider, **point-1.0**, which is the default value.

Query

```
CREATE POINT INDEX index_with_provider FOR (n:Label) ON (n.prop1) OPTIONS {indexProvider: 'point-1.0'}
```

Can be combined with specifying index configuration.

Result

```
| No data returned. |
| Indexes added: 1 |
```

15.13. Create a point index specifying the index configuration

To create a point index with a specific index configuration, the **OPTIONS** clause is used. The valid configuration settings are

- `spatial.cartesian.min`
- `spatial.cartesian.max`
- `spatial.cartesian-3d.min`
Non-specified settings have their respective default values.

Query

```
CREATE POINT INDEX index_with_config FOR (n:Label) ON (n.prop2)
OPTIONS {
indexConfig: {
'spatial.cartesian.min': [-100.0, -100.0],
'spatial.cartesian.max': [100.0, 100.0]
}}
```

Result

```
+-------------------+
<table>
<thead>
<tr>
<th>No data returned.</th>
</tr>
</thead>
</table>
Indexes added: 1
```

15.14. Create a point index specifying both the index provider and configuration

To create a point index with a specific index provider and configuration, the OPTIONS clause is used. Only one valid value exists for the index provider, point-1.0, which is the default value. The valid configuration settings are

- spatial.cartesian.min
- spatial.cartesian.max
- spatial.cartesian-3d.min
- spatial.cartesian-3d.max
- spatial.wgs-84.min
- spatial.wgs-84.max
- spatial.wgs-84-3d.min
- spatial.wgs-84-3d.max

Non-specified settings have their respective default values.

Query

```
CREATE POINT INDEX index_with_options FOR ()-[r:TYPE]->() ON (r.prop1)
OPTIONS {
indexProvider: 'point-1.0',
indexConfig: {
'spatial.wgs-84.min': [-100.0, -80.0],
'spatial.wgs-84.max': [100.0, 80.0]
}}
```
Specifying index provider and configuration can be done individually.

Result

<table>
<thead>
<tr>
<th>No data returned.</th>
</tr>
</thead>
</table>
Indexes added: 1
Chapter 16. Deprecated syntax

16.1. Drop a single-property index

A b-tree index on all nodes with 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
```

16.2. Drop a composite index

A composite b-tree index on all nodes with 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 an `age` and `country` property:

**Query**

```
DROP INDEX ON :Person(age, country)
```

**Result**

```
| No data returned. |
Indexes removed: 1
```
Chapter 17. Full-text search index

This chapter describes how to use full-text indexes, to enable full-text search.

Full-text indexes are powered by the Apache Lucene indexing and search library, and can be used to index nodes and relationships by string properties. A full-text index allows you to write queries that match within the contents of indexed string properties. For instance, the b-tree indexes described in previous sections can only do exact matching or prefix matches on strings. A full-text index will instead tokenize the indexed string values, so it can match terms anywhere within the strings. How the indexed strings are tokenized and broken into terms, is determined by what analyzer the full-text index is configured with. For instance, the swedish analyzer knows how to tokenize and stem Swedish words, and will avoid indexing Swedish stop words. The complete list of stop words for each analyzer is included in the result of the `db.index.fulltext.listAvailableAnalyzers` procedure.

Full-text indexes:

- support the indexing of both nodes and relationships.
- support configuring custom analyzers, including analyzers that are not included with Lucene itself.
- can be queried using the Lucene query language.
- can return the score for each result from a query.
- are kept up to date automatically, as nodes and relationships are added, removed, and modified.
- will automatically populate newly created indexes with the existing data in a store.
- can be checked by the consistency checker, and they can be rebuilt if there is a problem with them.
- are a projection of the store, and can only index nodes and relationships by the contents of their properties.
- can support any number of documents in a single index.
- are created, dropped, and updated transactionally, and is automatically replicated throughout a cluster.
- can be accessed via Cypher procedures.
- can be configured to be eventually consistent, in which index updating is moved from the commit path to a background thread. Using this feature, it is possible to work around the slow Lucene writes from the performance critical commit process, thus removing the main bottlenecks for Neo4j write performance.

At first sight, the construction of full-text indexes can seem similar to regular indexes. However there are some things that are interesting to note: In contrast to b-tree indexes, a full-text index can be:

- applied to more than one label.
- applied to more than one relationship type.
- applied to more than one property at a time (similar to a composite index) but with an important difference: While a composite index applies only to entities that match the indexed label and all of the indexed properties, full-text index will index entities that have at least one of the indexed labels or relationship types, and at least one of the indexed properties.
For information on how to configure full-text indexes, refer to Operations Manual → Indexes to support full-text search.

17.1. Full-text search procedures

Full-text indexes are managed through commands and used through built-in procedures, see Operations Manual → Procedures for a complete reference.

The commands and procedures for full-text indexes are listed in the table below:

<table>
<thead>
<tr>
<th>Usage</th>
<th>Procedure/Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create full-text node index</td>
<td>CREATE FULLTEXT INDEX ...</td>
<td>Create a node fulltext index for the given labels and properties. The optional 'options' map can be used to supply provider and settings to the index. Supported settings are 'fulltext.analyzer', for specifying what analyzer to use when indexing and querying. Use the db.index.fulltext.listAvailableAnalyzers procedure to see what options are available. And 'fulltext.eventually_consistent' which can be set to 'true' to make this index eventually consistent, such that updates from committing transactions are applied in a background thread.</td>
</tr>
<tr>
<td>Create full-text relationship index</td>
<td>CREATE FULLTEXT INDEX ...</td>
<td>Create a relationship fulltext index for the given relationship types and properties. The optional 'options' map can be used to supply provider and settings to the index. Supported settings are 'fulltext.analyzer', for specifying what analyzer to use when indexing and querying. Use the db.index.fulltext.listAvailableAnalyzers procedure to see what options are available. And 'fulltext.eventually_consistent' which can be set to 'true' to make this index eventually consistent, such that updates from committing transactions are applied in a background thread.</td>
</tr>
<tr>
<td>List available analyzers</td>
<td>db.index.fulltext.listAvailableAnalyzers</td>
<td>List the available analyzers that the full-text indexes can be configured with.</td>
</tr>
<tr>
<td>Use full-text node index</td>
<td>db.index.fulltext.queryNodes</td>
<td>Query the given full-text index. Returns the matching nodes and their Lucene query score, ordered by score.</td>
</tr>
<tr>
<td>Use full-text relationship index</td>
<td>db.index.fulltext.queryRelationships</td>
<td>Query the given full-text index. Returns the matching relationships and their Lucene query score, ordered by score.</td>
</tr>
</tbody>
</table>
### Usage
<table>
<thead>
<tr>
<th>Procedure/Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Drop full-text index</strong></td>
<td><strong>DROP INDEX ...</strong> Drop the specified index.</td>
</tr>
<tr>
<td><strong>Eventually consistent indexes</strong></td>
<td><strong>db.index.fulltext.awaitEventuallyConsistentIndexRefresh</strong> Wait for the updates from recently committed transactions to be applied to any eventually-consistent full-text indexes.</td>
</tr>
<tr>
<td><strong>Listing all fulltext indexes</strong></td>
<td><strong>SHOW FULLTEXT INDEXES</strong> Lists all fulltext indexes, see the <strong>SHOW INDEXES</strong> command for details.</td>
</tr>
</tbody>
</table>

## 17.2. Create and configure full-text indexes

Full-text indexes are created with the `CREATE FULLTEXT INDEX` command. An index can be given a unique name when created (or get a generated one), which is used to reference the specific index when querying or dropping it. A full-text index applies to a list of labels or a list of relationship types, for node and relationship indexes respectively, and then a list of property names.

*Table 448. Syntax for creating fulltext indexes*
**Command** | **Description** | **Comment**
---|---|---
```cypher
CREATE FULLTEXT INDEX [index_name] [IF NOT EXISTS]
FOR (n:LabelName[*] ...) 
ON EACH "[" n.propertyName[,...] "]"
[OPTIONS "{ "option: value[,...] " }"]
``` | Create a fulltext index on nodes. | Best practice is to give the index a name when it is created. This name is needed for both dropping and querying the index. If the index is not explicitly named, it will get an auto-generated name. The index name must be unique among all indexes and constraints. Index provider and configuration can be specified using the `OPTIONS` clause. The command is optionally idempotent, with the default behavior to throw an error if you attempt to create the same index twice. With `IF NOT EXISTS`, no error is thrown and nothing happens should an index with the same name, schema or both already exist. It may still throw an error should a constraint with the same name exist.

```cypher
CREATE FULLTEXT INDEX [index_name] [IF NOT EXISTS]
FOR ()-"[r:TYPE_NAME[*]...]"-() 
ON EACH "[" r.propertyName[,...] "]"
[OPTIONS "{ "option: value[,...] " }"]
``` | Create a fulltext index on relationships. | For instance, if we have a movie with a title.

**Query**

```cypher
CREATE (m:Movie {title: "The Matrix"}) RETURN m.title
```

**Table 449. Result**

<table>
<thead>
<tr>
<th>m.title</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;The Matrix&quot;</td>
</tr>
</tbody>
</table>

Rows: 1  
Nodes created: 1  
Properties set: 1  
Labels added: 1
And we have a full-text index on the **title** and **description** properties of movies and books.

Query

```sql
CREATE FULLTEXT INDEX titlesAndDescriptions FOR (n:Movie|Book) ON EACH [n.title, n.description]
```

Then our movie node from above will be included in the index, even though it only has one of the indexed labels, and only one of the indexed properties:

Query

```sql
CALL db.index.fulltext.queryNodes("titlesAndDescriptions", "matrix") YIELD node, score
RETURN node.title, node.description, score
```

Table 450. Result

<table>
<thead>
<tr>
<th>node.title</th>
<th>node.description</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;The Matrix&quot;</td>
<td>&lt;null&gt;</td>
<td>0.7799721956253052</td>
</tr>
</tbody>
</table>

Rows: 1

The same is true for full-text indexes on relationships. Though a relationship can only have one type, a relationship full-text index can index multiple types, and all relationships will be included that match one of the relationship types, and at least one of the indexed properties.

The **CREATE FULLTEXT INDEX** command take an optional clause, called **options**. This have two parts, the **indexProvider** and **indexConfig**. The provider can only have the default value, 'fulltext-1.0'. The **indexConfig** is a map from string to string and booleans, and can be used to set index-specific configuration settings. The **fulltext.analyzer** setting can be used to configure an index-specific analyzer. The possible values for the **fulltext.analyzer** setting can be listed with the **db.index.fulltext.listAvailableAnalyzers** procedure. The **fulltext.eventually_consistent** setting, if set to **true**, will put the index in an eventually consistent update mode. This means that updates will be applied in a background thread "as soon as possible", instead of during transaction commit like other indexes.

Query

```sql
CREATE FULLTEXT INDEX taggedByRelationshipIndex FOR ()-[r:TAGGED_AS]-() ON EACH [r.taggedByUser] OPTIONS {
  indexConfig: {
    'fulltext.analyzer': 'url_or_email',
    'fulltext.eventually_consistent': true
  }
}
```

In this example, an eventually consistent relationship full-text index is created for the **TAGGED_AS** relationship type, and the **taggedByUser** property, and the index uses the **url_or_email** analyzer. This could, for instance, be a system where people are assigning tags to documents, and where the index on the **taggedByUser** property will allow them to quickly find all of the documents they have tagged. Had it not been for the relationship index, one would have had to add artificial connective nodes between the tags and the documents in the data model, just so these nodes could be indexed.

Table 451. Result

(empty result)
17.3. Query full-text indexes

Full-text indexes will, in addition to any exact matches, also return approximate matches to a given query. Both the property values that are indexed, and the queries to the index, are processed through the analyzer such that the index can find that don't exactly matches. The score that is returned alongside each result entry, represents how well the index thinks that entry matches the given query. The results are always returned in descending score order, where the best matching result entry is put first. To illustrate, in the example below, we search our movie database for "Full Metal Jacket", and even though there is an exact match as the first result, we also get three other less interesting results:

**Query**

```
CALL db.index.fulltext.queryNodes("titlesAndDescriptions", "Full Metal Jacket") YIELD node, score
RETURN node.title, score
```

**Table 452. Result**

<table>
<thead>
<tr>
<th>node.title</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Full Metal Jacket&quot;</td>
<td>1.411118507385254</td>
</tr>
<tr>
<td>&quot;Full Moon High&quot;</td>
<td>0.44524085521698</td>
</tr>
<tr>
<td>&quot;Yellow Jacket&quot;</td>
<td>0.3509605824947357</td>
</tr>
<tr>
<td>&quot;The Jacket&quot;</td>
<td>0.3509605824947357</td>
</tr>
</tbody>
</table>

Rows: 4

Full-text indexes are powered by the Apache Lucene indexing and search library. This means that we can use Lucene’s full-text query language to express what we wish to search for. For instance, if we are only interested in exact matches, then we can quote the string we are searching for.

**Query**

```
CALL db.index.fulltext.queryNodes("titlesAndDescriptions", "Full Metal Jacket") YIELD node, score
RETURN node.title, score
```

When we put "Full Metal Jacket" in quotes, Lucene only gives us exact matches.

**Table 453. Result**

<table>
<thead>
<tr>
<th>node.title</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Full Metal Jacket&quot;</td>
<td>1.411118507385254</td>
</tr>
</tbody>
</table>

Rows: 1

Lucene also allows us to use logical operators, such as AND and OR, to search for terms:
CALL db.index.fulltext.queryNodes("titlesAndDescriptions", 'full AND metal') YIELD node, score
RETURN node.title, score

Only the **Full Metal Jacket** movie in our database has both the words *full* and *metal*.

Table 454. Result

<table>
<thead>
<tr>
<th>node.title</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Full Metal Jacket&quot;</td>
<td>1.1113792657852173</td>
</tr>
</tbody>
</table>

Rows: 1

It is also possible to search for only specific properties, by putting the property name and a colon in front of the text being searched for.

Query

CALL db.index.fulltext.queryNodes("titlesAndDescriptions", 'description:"surreal adventure"') YIELD node, score
RETURN node.title, node.description, score

Table 455. Result

<table>
<thead>
<tr>
<th>node.title</th>
<th>node.description</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Metallica Through The Never&quot;</td>
<td>&quot;The movie follows the young roadie Trip through his surreal adventure with the band.&quot;</td>
<td>0.2615291476249695</td>
</tr>
</tbody>
</table>

Rows: 1

A complete description of the Lucene query syntax can be found in the [Lucene documentation](#).

### 17.4. Drop full-text indexes

A full-text node index is dropped by using the **same command as for other indexes**, `DROP INDEX`.

In the following example, we will drop the `taggedByRelationshipIndex` that we created previously:

Query

DROP INDEX taggedByRelationshipIndex

Table 456. Result

(emptys result)

Rows: 0
Indexes removed: 1
Chapter 18. Constraints

This section explains how to manage constraints used for ensuring data integrity.

18.1. Types of constraint

The following constraint types are available:

**Unique node property constraints**

Unique property constraints ensure that property values are unique for all nodes with a specific label. For unique property constraints on multiple properties, the combination of the property values is unique. Unique constraints do not require all nodes to have a unique value for the properties listed — nodes without all properties are not subject to this rule.

**Node property existence constraints**

Node property existence constraints ensure that a property exists for all nodes with a specific label. Queries that try to create new nodes of the specified label, but without this property, will fail. The same is true for queries that try to remove the mandatory property.

**Relationship property existence constraints**

Property existence constraints ensure that a property exists for all relationships with a specific type. All queries that try to create relationships of the specified type, but without this property, will fail. The same is true for queries that try to remove the mandatory property.

**Node key constraints**

Node key constraints 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.

---

Node key constraints, node property existence constraints and relationship property existence constraints are only available in Neo4j Enterprise Edition. Databases containing one of these constraint types cannot be opened using Neo4j Community Edition.
18.2. Implications on indexes

Creating a constraint has the following implications on indexes:

- Adding a node key or unique property constraint on a single property also adds an index on that property and therefore, an index of the same index type, label, and property combination cannot be added separately.

- Adding a node key or unique property constraint for a set of properties also adds an index on those properties and therefore, an index of the same index type, label, and properties combination cannot be added separately.

- Cypher will use these indexes for lookups just like other indexes. Refer to Indexes for search performance for more details on indexes.

- If a node key or unique property constraint is dropped and the backing index is still required, the index need to be created explicitly.

Additionally, the following is true for constraints:

- 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'.

- Best practice is to give the constraint a name when it is created. If the constraint is not explicitly named, it will get an auto-generated name.

- The constraint name must be unique among both indexes and constraints.

- Constraint creation is by default not idempotent, and an error will be thrown if you attempt to create the same constraint twice. Using the keyword IF NOT EXISTS makes the command idempotent, and no error will be thrown if you attempt to create the same constraint twice.

18.3. Syntax

Syntax for how to manage constraints used for ensuring data integrity.

18.3.1. Syntax for creating constraints

Best practice when creating a constraint is to give the constraint a name. This name must be unique among both indexes and constraints. If a name is not explicitly given, a unique name will be auto-generated.

The create constraint command is optionally idempotent, with the default behavior to throw an error if you attempt to create the same constraint twice. With the IF NOT EXISTS flag, no error is thrown and nothing happens should a constraint with the same name or same schema and constraint type already exist. It may still throw an error if conflicting data, indexes, or constraints exist. Examples of this are nodes with missing properties, indexes with the same name, or constraints with same schema but a different constraint type.
For constraints that are backed by an index, the index provider and configuration for the backing index can be specified using the `OPTIONS` clause. Valid values for the index provider are `native-btree-1.0` (deprecated), `lucene+native-3.0` (deprecated), and `range-1.0` (future index), default is `native-btree-1.0`. The index type of the backing index is set depending on the provider, the `range-1.0` generates a future range index while the other providers generates a b-tree index. The range index has no configuration settings. The valid b-tree configuration settings are

- `spatial.cartesian.min`
- `spatial.cartesian.max`
- `spatial.cartesian-3d.min`
- `spatial.cartesian-3d.max`
- `spatial.wgs-84.min`
- `spatial.wgs-84.max`
- `spatial.wgs-84-3d.min`
- `spatial.wgs-84-3d.max`

Non-specified settings have their respective default values.

Creating a constraint requires the `CREATE CONSTRAINT` privilege.

### Create a unique node property constraint

This command creates a uniqueness constraint on nodes with the specified label and properties.

```
CREATE CONSTRAINT [constraint_name] [IF NOT EXISTS]
FOR (n:LabelName)
REQUIRE n.propertyName IS UNIQUE
[OPTIONS "{" option: value[, ... ] "}"
```

Index provider and configuration can be specified using the `OPTIONS` clause.

### Create a node property existence constraint

This command creates a property existence constraint on nodes with the specified label and property.

```
CREATE CONSTRAINT [constraint_name] [IF NOT EXISTS]
FOR (n:LabelName)
REQUIRE n.propertyName IS NOT NULL
[OPTIONS "{""]
```

There are no supported `OPTIONS` values for existence constraints, but an empty options map is allowed for consistency.
Create a relationship property existence constraint

This command creates a property existence constraint on relationships with the specified relationship type and property.

```
CREATE CONSTRAINT [constraint_name] [IF NOT EXISTS] FOR ()->["r:RELATIONSHIP_TYPE"]-() REQUIRE r.propertyName IS NOT NULL [OPTIONS "{}"]
```

There are no supported OPTIONS values for existence constraints, but an empty options map is allowed for consistency.

Create a node key constraint

This command creates a node key constraint on nodes with the specified label and properties.

```
CREATE CONSTRAINT [constraint_name] [IF NOT EXISTS] FOR (n:LabelName) REQUIRE n.propertyName IS NODE KEY [OPTIONS "{"option: value[, ...] "}"]
```

```
CREATE CONSTRAINT [constraint_name] [IF NOT EXISTS] FOR (n:LabelName) REQUIRE (n.propertyName_1, ..., n.propertyName_n) IS NODE KEY [OPTIONS "{"option: value[, ...] "}"]
```

Index provider and configuration can be specified using the OPTIONS clause.

18.3.2. Syntax for dropping constraints

Drop a constraint

The preferred way of dropping a constraint is by the name of the constraint.

This drop command is optionally idempotent, with the default behavior to throw an error if you attempt to drop the same constraint twice. With the IF EXISTS flag, no error is thrown and nothing happens should the constraint not exist.

Dropping a constraint requires the DROP CONSTRAINT privilege.

```
DROP CONSTRAINT constraint_name [IF EXISTS]
```

Drop a unique constraint without specifying a name

An old way of dropping a uniqueness constraint was to drop the constraint by specifying the schema of the constraint.
Drop a node property existence constraint without specifying a name

An old way of dropping a node property existence constraint was to drop the constraint by specifying the schema of the constraint.

Drop a relationship property existence constraint without specifying a name

An old way of dropping a relationship property existence constraint was to drop the constraint by specifying the schema of the constraint.

Drop a node key constraint without specifying a name

An old way of dropping a node key constraint was to drop the constraint by specifying the schema of the constraint.

18.3.3. Syntax for listing constraints

List constraints in the database, either all or filtered on constraint type. This requires the \texttt{SHOW CONSTRAINT} privilege.

The simple version of the command allows for a \texttt{WHERE} clause and will give back the default set of output columns:
To get the full set of output columns, a yield clause is needed:

```sql
WHERE expression

YIELD { * | field[, ...] } [ORDER BY field[, ...]] [SKIP n] [LIMIT n]
WHERE expression
RETURN field[, ...] [ORDER BY field[, ...]] [SKIP n] [LIMIT n]
```

The returned columns from the show command is:

Table 457. List constraints output

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Default output</th>
<th>Full output</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The id of the constraint.</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>name</td>
<td>Name of the constraint (explicitly set by the user or automatically assigned).</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>type</td>
<td>The ConstraintType of this constraint (UNIQUENESS, NODE_PROPERTY_EXISTENCE, NODE_KEY, or RELATIONSHIP_PROPERTY_EXISTENCE).</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>entityType</td>
<td>Type of entities this constraint represents (nodes or relationship).</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>labelsOrTypes</td>
<td>The labels or relationship types of this constraint.</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>properties</td>
<td>The properties of this constraint.</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ownedIndexId</td>
<td>The id of the index associated to the constraint, or null if no index is associated with the constraint.</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>options</td>
<td>The options passed to <code>CREATE</code> command, for the index associated to the constraint, or null if no index is associated with the constraint.</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>createStatement</td>
<td>Statement used to create the constraint.</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

The deprecated built-in procedures for listing constraints, such as `db.constraints`, work as before and are not affected by the `SHOW CONSTRAINTS` privilege.
18.4. Examples

Examples of how to manage constraints used for ensuring data integrity.

18.4.1. Unique node property constraints

Create a unique constraint

When creating a unique constraint, a name can be provided. The constraint ensures that your database will never contain more than one node with a specific label and one property value.

Query

```
CREATE CONSTRAINT constraint_name FOR (book:Book) REQUIRE book.isbn IS UNIQUE
```

Result

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

Unique constraints added: 1
```

Create a unique constraint only if it does not already exist

If it is not known whether a constraint exists or not, add `IF NOT EXISTS` to ensure it does. The uniqueness constraint ensures that your database will never contain more than one node with a specific label and one property value.

Query

```
CREATE CONSTRAINT constraint_name IF NOT EXISTS FOR (book:Book) REQUIRE book.isbn IS UNIQUE
```

Note no constraint will be created if any other constraint with that name or another uniqueness constraint on the same schema already exists. Assuming no such constraints existed:

Result

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

Unique constraints added: 1
```

Create a unique constraint with specified index provider and configuration

To create a unique constraint with a specific index provider and configuration for the backing index, the `OPTIONS` clause is used. Valid values for the index provider are `native-btree-1.0` (deprecated), `lucene+native-3.0` (deprecated), and `range-1.0` (future index), default is `native-btree-1.0`. The index type
of the backing index is set depending on the provider, the `range-1.0` generates a future range index while the other providers generates a b-tree index. The range index have no configuration settings. The valid b-tree configuration settings are

- `spatial.cartesian.min`
- `spatial.cartesian.max`
- `spatial.cartesian-3d.min`
- `spatial.cartesian-3d.max`
- `spatial.wgs-84.min`
- `spatial.wgs-84.max`
- `spatial.wgs-84-3d.min`
- `spatial.wgs-84-3d.max`

Non-specified settings have their respective default values.

In 4.4, b-tree index-backed constraints are still the correct alternative to use.

Query

```plaintext
CREATE CONSTRAINT constraint_with_options FOR (n:Label) REQUIRE (n.prop1, n.prop2) IS UNIQUE OPTIONS {
    indexProvider: 'lucene+native-3.0',
    indexConfig: {
        'spatial.wgs-84.min': [-100.0, -80.0],
        'spatial.wgs-84.max': [100.0, 80.0]
    }
}
```

Specifying index provider and configuration can be done individually.

Result

```
+-------------------+
| No data returned. |
+-------------------+
Unique constraints added: 1
```

Failure to create an already existing unique property constraint

Create a unique property constraint on the property `title` on nodes with the `Book` label, when that constraint already exists.

Query

```plaintext
CREATE CONSTRAINT FOR (book:Book) REQUIRE book.title IS UNIQUE
```

In this case the constraint can't be created because it already exists.

Error message

```
Constraint already exists: Constraint( id=4, name='preExistingUnique',
type='UNIQUENESS', schema!(:Book {title}), ownedIndex=3 )
```
Failure to create a unique property constraint on same schema as existing index

Create a unique property constraint on the property `wordCount` on nodes with the `Book` label, when an index already exists on that label and property combination.

Query

```
CREATE CONSTRAINT FOR (book:Book) REQUIRE book.wordCount IS UNIQUE
```

In this case the constraint can’t be created because there already exists an index covering that schema.

Error message

```
There already exists an index (:Book {wordCount}). A constraint cannot be created until the index has been dropped.
```

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
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 FOR (book:Book) REQUIRE 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 for search performance instead or remove the offending nodes and then re-apply the constraint.

Error message

```
Unable to create Constraint( name='constraint_62365a16', type='UNIQUENESS',
schema=(:Book {isbn})):
Both Node(0) and Node(1) have the label `Book` and property `isbn` = '1449356265'
```
Chapter 19. Node property existence constraints

Create a node property existence constraint

When creating a node property existence constraint, a name can be provided. The constraint ensures that all nodes with a certain label have a certain property.

Query

```
CREATE CONSTRAINT constraint_name FOR (book:Book) REQUIRE book.isbn IS NOT NULL
```

Result

```
+-------------------+
| No data returned. |
+-------------------+
Property existence constraints added: 1
```

Create a node property existence constraint only if it does not already exist

If it is not known whether a constraint exists or not, add IF NOT EXISTS to ensure it does. The node property existence constraint ensures that all nodes with a certain label have a certain property.

Query

```
CREATE CONSTRAINT constraint_name IF NOT EXISTS FOR (book:Book) REQUIRE book.isbn IS NOT NULL
```

Note no constraint will be created if any other constraint with that name or another node property existence constraint on the same schema already exists. Assuming a constraint with the name `constraint_name` already existed:

Result

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

Failure to create an already existing node property existence constraint

Create a node property existence constraint on the property `title` on nodes with the `Book` label, when that constraint already exists.
Query

```sql
CREATE CONSTRAINT booksShouldHaveTitles FOR (book:Book) REQUIRE book.title IS NOT NULL
```

In this case the constraint can't be created because it already exists.

Error message

```sql
Constraint already exists: Constraint( id=3, name='preExistingNodePropExist', type='NODE PROPERTY EXISTENCE', schema=(:Book {title}) )
```

Create a node that complies with property existence constraints

Create a `Book` node with an `isbn` property.

Query

```sql
CREATE (book:Book {isbn: '1449356265', title: 'Graph Databases'})
```

Result

```
<table>
<thead>
<tr>
<th>No data returned.</th>
</tr>
</thead>
</table>
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

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

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

Error message

```sql
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 FOR (book:Book) REQUIRE book.isbn IS NOT NULL
```

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( type='NODE PROPERTY EXISTENCE', schema=(:Book {isbn})): Node(0) with label `Book` must have the property `isbn`
```
Chapter 20. Relationship property existence constraints

Enterprise edition

Create a relationship property existence constraint

When creating a relationship property existence constraint, a name can be provided. The constraint ensures all relationships with a certain type have a certain property.

Query

```
CREATE CONSTRAINT constraint_name FOR ()-[like:LIKED]-() REQUIRE like.day IS NOT NULL
```

Result

```
+-------------------+
| No data returned. |
+-------------------+
Property existence constraints added: 1
```

Create a relationship property existence constraint only if it does not already exist

If it is not known whether a constraint exists or not, add IF NOT EXISTS to ensure it does. The relationship property existence constraint ensures all relationships with a certain type have a certain property.

Query

```
CREATE CONSTRAINT constraint_name IF NOT EXISTS FOR ()-[like:LIKED]-() REQUIRE like.day IS NOT NULL
```

Note no constraint will be created if any other constraint with that name or another relationship property existence constraint on the same schema already exists. Assuming a constraint with the name `constraint_name` already existed:

Result

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

Failure to create an already existing relationship property existence constraint

Create a named relationship property existence constraint on the property `week` on relationships with the `LIKED` type, when a constraint with that name already exists.
Query

CREATE CONSTRAINT relPropExist FOR ()-[like:LIKED]-() REQUIRE like.week IS NOT NULL

In this case the constraint can’t be created because there already exists a constraint with that name.

Error message

There already exists a constraint called 'relPropExis'.

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 FOR ()-[like:LIKED]-() REQUIRE like.day IS NOT NULL

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( type='RELATIONSHIP PROPERTY EXISTENCE',
  schema=-[:LIKED {day}]- ): Relationship(0) with type `LIKED` must have the property `day'
Chapter 21. Node key constraints

Create a node key constraint

When creating a node key constraint, a name can be provided. The constraint ensures that all nodes with a particular label have a set of defined properties whose combined value is unique and all properties in the set are present.

Query

```
CREATE CONSTRAINT constraint_name FOR (n:Person) REQUIRE (n.firstname, n.surname) IS NODE KEY
```

Result

```
+-------------------+
| No data returned. |
+-------------------+
Node key constraints added: 1
```

Create a node key constraint only if it does not already exist

If it is not known whether a constraint exists or not, add `IF NOT EXISTS` to ensure it does. The node key constraint ensures that all nodes with a particular label have a set of defined properties whose combined value is unique and all properties in the set are present.

Query

```
CREATE CONSTRAINT constraint_name IF NOT EXISTS FOR (n:Person) REQUIRE (n.firstname, n.surname) IS NODE KEY
```

Note no constraint will be created if any other constraint with that name or another node key constraint on the same schema already exists. Assuming a node key constraint on `(:Person {firstname, surname})` already existed:

Result

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

Create a node key constraint with specified index provider

To create a node key constraint with a specific index provider for the backing index, the `OPTIONS` clause is used. Valid values for the index provider are `native-btree-1.0` (deprecated), `lucene+native-3.0` (deprecated), and `range-1.0` (future index), default is `native-btree-1.0`. The index type of the backing index is set depending on the provider, the `range-1.0` generates a future range index while the other providers generates a b-tree index. In 4.4, b-tree index-backed constraints are still the correct alternative.
to use.

Query

```plaintext
CREATE CONSTRAINT constraint_with_provider FOR (n:Label) REQUIRE (n.prop1) IS NODE KEY OPTIONS (indexProvider: 'native-btree-1.0')
```

B-tree providers can be combined with specifying index configuration.

Result

```
| No data returned. |
+-------------------+
Node key constraints added: 1
```

Create a node key constraint with specified index configuration

To create a node key constraint with a specific index configuration for the backing index, the `OPTIONS` clause is used. The index type of the backing index is set depending on the provider and future range indexes have no configuration settings. The valid b-tree configuration settings are

- `spatial.cartesian.min`
- `spatial.cartesian.max`
- `spatial.cartesian-3d.min`
- `spatial.cartesian-3d.max`
- `spatial.wgs-84.min`
- `spatial.wgs-84.max`
- `spatial.wgs-84-3d.min`
- `spatial.wgs-84-3d.max`

Non-specified settings have their respective default values.

Query

```plaintext
CREATE CONSTRAINT constraint_with_config FOR (n:Label) REQUIRE (n.prop2) IS NODE KEY OPTIONS (indexConfig: {'spatial.cartesian.min': [-100.0, -100.0], 'spatial.cartesian.max': [100.0, 100.0]})
```

Can be combined with specifying a b-tree index provider.

Result

```
| No data returned. |
+-------------------+
Node key constraints added: 1
```

Failure to create a node key constraint when a unique property
constraint exists on the same schema

Create a node key constraint on the properties `firstname` and `age` on nodes with the `Person` label, when a unique property constraint already exists on the same label and property combination.

Query

```
CREATE CONSTRAINT FOR (p:Person) REQUIRE (p.firstname, p.age) IS NODE KEY
```

In this case the constraint can’t be created because there already exist a conflicting constraint on that label and property combination.

Error message

```
Constraint already exists: Constraint( id=4, name='preExistingUnique',
type='UNIQUENESS', schema=(:Person {firstname, age}), ownedIndex=3 )
```

Failure to create a node key constraint with the same name as existing index

Create a named node key constraint on the property `title` on nodes with the `Book` label, when an index already exists with that name.

Query

```
CREATE CONSTRAINT bookTitle FOR (book:Book) REQUIRE book.title IS NODE KEY
```

In this case the constraint can’t be created because there already exists an index with that name.

Error message

```
There already exists an index called 'bookTitle'.
```

Create a node that complies with node key constraints

Create a `Person` node with both a `firstname` and `surname` property.

Query

```
CREATE (p:Person {firstname: 'John', surname: 'Wood', age: 55})
```

Result

```
+-------------------+
<table>
<thead>
<tr>
<th>No data returned.</th>
</tr>
</thead>
</table>
+-------------------+
Nodes created: 1
Properties set: 3
Labels added: 1
```
Create a node that violates a node key constraint

Trying to create a `Person` node without a `surname` property, given a node key constraint 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)
```

Removing a NODE KEY-constrained property

Trying to remove the `surname` property from an existing node `Person`, given a NODE KEY constraint on `:Person(firstname, surname)`.

Query

```
MATCH (p:Person {firstname: 'John', surname: 'Wood'}) REMOVE p.surname
```

In this case the property is not removed.

Error message

```
Node(0) with label `Person` must have the properties (firstname, surname)
```

Failure to create a node key constraint due to existing node

Trying to create a node key constraint 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 FOR (n:Person) REQUIRE (n.firstname, n.surname) IS NODE KEY
```

In this case the node key 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( name='constraint_4020f37a', type='NODE KEY', schema=(:Person {firstname, surname}) ): Failed during property existence validation: Unable to create constraint Constraint( type='NODE PROPERTY EXISTENCE', schema=(:Label[0] {PropertyKey[10], PropertyKey[25]}): Node(0) does not satisfy Constraint( type='NODE PROPERTY EXISTENCE', schema=(:Person {firstname, surname}) ).
```
Chapter 22. Drop a constraint by name

Drop a constraint

A constraint can be dropped using the name with the `DROP CONSTRAINT constraint_name` command. It is the same command for unique property, property existence and node key constraints. The name of the constraint can be found using the `SHOW CONSTRAINTS` command, given in the output column `name`.

Query

```
DROP CONSTRAINT constraint_name
```

Result

```
+-------------------+
| No data returned. |
+-------------------+
Named constraints removed: 1
```

Drop a non-existing constraint

If it is uncertain if any constraint with a given name exists and you want to drop it if it does but not get an error should it not, use `IF EXISTS`. It is the same command for unique property, property existence and node key constraints.

Query

```
DROP CONSTRAINT missing_constraint_name IF EXISTS
```

Result

```
+----------------------------------------------------------+
| No data returned, and nothing was changed.                 |
+----------------------------------------------------------+
```
Chapter 23. Listing constraints

Listing all constraints

To list all constraints with the default output columns, the `SHOW CONSTRAINTS` command can be used. If all columns are required, use `SHOW CONSTRAINTS YIELD *`.

Query

```
SHOW CONSTRAINTS
```

One of the output columns from `SHOW CONSTRAINTS` is the name of the constraint. This can be used to drop the constraint with the `DROP CONSTRAINT` command.

Result

```
+----------------------------------------------------------------------------------------------------+
| id | name                  | type         | entityType | labelsOrTypes | properties | ownedIndexId |
+----------------------------------------------------------------------------------------------------+
| 4  | "constraint_62365a16" | "UNIQUENESS" | "NODE"     | ["Book"]      | ["isbn"]   | 3           |
+----------------------------------------------------------------------------------------------------+
1 row
```

Listing constraints with filtering

One way of filtering the output from `SHOW CONSTRAINTS` by constraint type is the use of type keywords, listed in `Syntax for listing constraints`. For example, to show only unique node property constraints, use `SHOW UNIQUE CONSTRAINTS`. Another more flexible way of filtering the output is to use the `WHERE` clause. An example is to only show constraints on relationships.

Query

```
SHOW EXISTENCE CONSTRAINTS WHERE entityType = 'RELATIONSHIP'
```

This will only return the default output columns. To get all columns, use `SHOW INDEXES YIELD * WHERE ...`

Result

```
+---------------------------------------------------------------------------------------------------------+-------------------------------------------------------------------------------------------------------+-------------------------------------------------------------------------------------------------------+----------------------------------------------------------------------------------------------------------------+---------------------------------------------------------------------------------------------------------+-------------------------------------------------------------------------------------------------------+
| | name                  | type                                      | entityType     | labelsOrTypes | properties | ownedIndexId |
+---------------------------------------------------------------------------------------------------------+-------------------------------------------------------------------------------------------------------+-------------------------------------------------------------------------------------------------------+----------------------------------------------------------------------------------------------------------------+---------------------------------------------------------------------------------------------------------+-------------------------------------------------------------------------------------------------------+
| 7  | "constraint_f076a74d" | "RELATIONSHIP_PROPERTY_EXISTENCE" | "RELATIONSHIP" | ["KNOWS"] | ["since"] | <null>       |
+---------------------------------------------------------------------------------------------------------+-------------------------------------------------------------------------------------------------------+-------------------------------------------------------------------------------------------------------+----------------------------------------------------------------------------------------------------------------+---------------------------------------------------------------------------------------------------------+-------------------------------------------------------------------------------------------------------+
1 row
```
Chapter 24. Deprecated syntax

Create a unique constraint using deprecated syntax

The unique constraint ensures that your database will never contain more than one node with a specific label and one property value.

Query

```
CREATE CONSTRAINT ON (book:Book) ASSERT book.title IS UNIQUE
```

Result

```
+-------------------+
| No data returned. |
+-------------------+
Unique constraints added: 1
```

Drop a unique constraint

By using `DROP CONSTRAINT`, a b-tree index-backed unique constraint is removed 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 property existence constraint using deprecated syntax

The node property existence constraint ensures that all nodes with a certain label have a certain property.

Query

```
CREATE CONSTRAINT ON (book:Book) ASSERT book.title IS NOT NULL
```

Result

```
+-------------------+
| No data returned. |
+-------------------+
Property existence constraints added: 1
```
Create a node property existence constraint using deprecated syntax 2

The node property existence constraint ensures that all nodes with a certain label have a certain property.

Query

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

Result

+-------------------+
| No data returned. |
+-------------------+
Property existence constraints added: 1

Drop a node property existence constraint

By using `DROP CONSTRAINT`, a node property existence constraint is removed from the database.

Query

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

Result

+-------------------+
| No data returned. |
+-------------------+
Property existence constraints removed: 1

Create a relationship property existence constraint using deprecated syntax 1

The relationship property existence constraint ensures all relationships with a certain type have a certain property.

Query

```sql
CREATE CONSTRAINT ON ()-[like:LIKED]-() ASSERT like.week IS NOT NULL
```

Result

+-------------------+
| No data returned. |
+-------------------+
Property existence constraints added: 1

Create a relationship property existence constraint using
The relationship property existence constraint ensures all relationships with a certain type have a certain property.

**Query**

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

**Result**

```
<table>
<thead>
<tr>
<th>No data returned.</th>
</tr>
</thead>
</table>
Property existence constraints added: 1
```

---

**Drop a relationship property existence constraint**

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

**Query**

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

**Result**

```
<table>
<thead>
<tr>
<th>No data returned.</th>
</tr>
</thead>
</table>
Property existence constraints removed: 1
```

---

**Create a node key constraint using deprecated syntax**

The node key constraint ensures that all nodes with a particular label have a set of defined properties whose combined value is unique and all properties in the set are present.

**Query**

```
CREATE CONSTRAINT ON (n:Person) ASSERT (n.firstname) IS NODE KEY
```

**Result**

```
<table>
<thead>
<tr>
<th>No data returned.</th>
</tr>
</thead>
</table>
Node key constraints added: 1
```

---

**Drop a node key constraint**

Use `DROP CONSTRAINT` to remove a b-tree index-backed node key constraint 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
```
Chapter 25. Database management

This chapter explains how to use Cypher to manage Neo4j databases: creating, modifying, deleting, starting, and stopping individual databases within a single server.

Neo4j supports the management of multiple databases within the same DBMS. The metadata for these databases, including the associated security model, is maintained in a special database called the system database. All multi-database administrative commands must be run against the system database. These administrative commands are automatically routed to the system database when connected to the DBMS over Bolt.

The syntax of the database management commands is as follows:

Table 458. Database management command syntax

<table>
<thead>
<tr>
<th>Command</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Show Database</td>
<td>SHOW { DATABASE name</td>
</tr>
<tr>
<td>Show Database</td>
<td>SHOW { DATABASE name</td>
</tr>
<tr>
<td>Create Database</td>
<td>CREATE DATABASE name [IF NOT EXISTS] [OPTIONS &quot;{&quot; option: value[,...] &quot;}]* [WAIT [n [SEC[OND[S]]]]</td>
</tr>
<tr>
<td>Create Database</td>
<td>CREATE OR REPLACE DATABASE name [WAIT [n [SEC[OND[S]]]]</td>
</tr>
<tr>
<td>Alter Database</td>
<td>ALTER DATABASE name [IF EXISTS] SET ACCESS {READ ONLY</td>
</tr>
<tr>
<td>Stop Database</td>
<td>STOP DATABASE name [WAIT [n [SEC[OND[S]]]]</td>
</tr>
<tr>
<td>Start Database</td>
<td>START DATABASE name [WAIT [n [SEC[OND[S]]]]</td>
</tr>
<tr>
<td>Drop Database</td>
<td>DROP DATABASE name [IF EXISTS] [[DUMP</td>
</tr>
</tbody>
</table>
25.1. Listing databases

There are four different commands for listing databases: listing all databases, listing a particular database, listing the default database, and listing the home database. These commands return the following columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Default Output</th>
<th>Full Output</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name of the database.</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>aliases</td>
<td>The names of any aliases the database may have.</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>access</td>
<td>The database access mode, either read-write or read-only.</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>databaseID</td>
<td>The database unique ID.</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>serverID</td>
<td>The server instance ID.</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>address</td>
<td>Instance address in a clustered DBMS. The default for a standalone database is neo4j://localhost:7687.</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>role</td>
<td>The current role of the database (standalone, leader, follower, read_replica, unknown).</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>requestedStatus</td>
<td>The expected status of the database.</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>currentStatus</td>
<td>The actual status of the database.</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>error</td>
<td>An error message explaining why the database is not in the correct state.</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>default</td>
<td>Show if this is the default database for the DBMS.</td>
<td>✓</td>
<td>✓</td>
<td>Not returned by SHOW HOME DATABASE or SHOW DEFAULT DATABASE.</td>
</tr>
<tr>
<td>home</td>
<td>Shown if this is the home database for the current user.</td>
<td>✓</td>
<td>✓</td>
<td>Not returned by SHOW HOME DATABASE or SHOW DEFAULT DATABASE.</td>
</tr>
<tr>
<td>lastCommittedTxn</td>
<td>The ID of the last transaction received.</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>replicationLag</td>
<td>Number of transactions the current database is behind compared to the database on the primary instance. The lag is expressed in negative integers. In standalone environments, the value is always 0.</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A summary of all available databases can be displayed using the command `SHOW DATABASES`.

Query

```sql
SHOW DATABASES
```

Table 459. Result
Note that the results of this command are filtered according to the ACCESS privileges of the user. However, users with CREATE/DROP/ALTER DATABASE, SET DATABASE ACCESS, or DATABASE MANAGEMENT privileges can see all databases regardless of their ACCESS privileges. If a user has not been granted ACCESS privilege to any databases, the command can still be executed but will only return the system database, which is always visible.

In this example, the detailed information for a particular database can be displayed using the command SHOW DATABASE name YIELD *. When a YIELD clause is provided, the full set of columns is returned.

Query

```
SHOW DATABASE movies YIELD *
```

Table 460. Result

<table>
<thead>
<tr>
<th>name</th>
<th>aliases</th>
<th>access</th>
<th>address</th>
<th>role</th>
<th>requested Status</th>
<th>current Status</th>
<th>error</th>
<th>default</th>
<th>home</th>
<th>lastCommittedTxn</th>
<th>replicationLag</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;movies&quot;</td>
<td>[&quot;films&quot;,&quot;motion pictures&quot;]</td>
<td>&quot;read-write&quot;</td>
<td>localhost:7687</td>
<td>&quot;standalone&quot;</td>
<td>&quot;online&quot;</td>
<td>&quot;online&quot;</td>
<td>&quot;&quot;</td>
<td>false</td>
<td>false</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Rows: 1

The number of databases can be seen using a count() aggregation with YIELD and RETURN.

Query

```
SHOW DATABASES YIELD * RETURN count(*) as count
```

Table 461. Result
The default database can be seen using the command `SHOW DEFAULT DATABASE`.

Query

```
SHOW DEFAULT DATABASE
```

Table 462. Result

<table>
<thead>
<tr>
<th>name</th>
<th>aliases</th>
<th>access</th>
<th>address</th>
<th>role</th>
<th>requestedStatus</th>
<th>currentStatus</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;neo4j&quot;</td>
<td>[]</td>
<td>&quot;read-write&quot;</td>
<td>&quot;localhost:7687&quot;</td>
<td>&quot;standalone&quot;</td>
<td>&quot;online&quot;</td>
<td>&quot;online&quot;</td>
<td>&quot;&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

The home database for the current user can be seen using the command `SHOW HOME DATABASE`.

Query

```
SHOW HOME DATABASE
```

Table 463. Result

<table>
<thead>
<tr>
<th>name</th>
<th>aliases</th>
<th>access</th>
<th>address</th>
<th>role</th>
<th>requestedStatus</th>
<th>currentStatus</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;neo4j&quot;</td>
<td>[]</td>
<td>&quot;read-write&quot;</td>
<td>&quot;localhost:7687&quot;</td>
<td>&quot;standalone&quot;</td>
<td>&quot;online&quot;</td>
<td>&quot;online&quot;</td>
<td>&quot;&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

It is also possible to filter and sort the results by using `YIELD, ORDER BY and WHERE`.

Query

```
SHOW DATABASES YIELD name, currentStatus, requestedStatus ORDER BY currentStatus WHERE name CONTAINS 'e'
```

In this example:

- The number of columns returned has been reduced with the `YIELD` clause.
- The order of the returned columns has been changed.
- The results have been filtered to only show database names containing 'e'.
- The results are ordered by the 'currentStatus' column using `ORDER BY`.

It is also possible to use `SKIP` and `LIMIT` to paginate the results.

Table 464. Result
<table>
<thead>
<tr>
<th>name</th>
<th>currentStatus</th>
<th>requestedStatus</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;movies&quot;</td>
<td>&quot;online&quot;</td>
<td>&quot;online&quot;</td>
</tr>
<tr>
<td>&quot;neo4j&quot;</td>
<td>&quot;online&quot;</td>
<td>&quot;online&quot;</td>
</tr>
<tr>
<td>&quot;system&quot;</td>
<td>&quot;online&quot;</td>
<td>&quot;online&quot;</td>
</tr>
</tbody>
</table>

Note that for failed databases, the **currentStatus** and **requestedStatus** are different. This often implies an error, but does not always. For example, a database may take a while to transition from **offline** to **online** due to performing recovery. Or, during normal operation a database’s **currentStatus** may be transiently different from its **requestedStatus** due to a necessary automatic process, such as one Neo4j instance copying store files from another. The possible statuses are **initial**, **online**, **offline**, **store copying**, and **unknown**.

### 25.2. Creating databases

**Enterprise edition**

Databases can be created using `CREATE DATABASE`.

**Query**

```
CREATE DATABASE customers
```

0 rows, System updates: 1

Database names are subject to the standard Cypher restrictions on valid identifiers. The following naming rules apply:

- Database name length must be between 3 and 63 characters.
- The first character must be an ASCII alphabetic character.
- Subsequent characters can be ASCII alphabetic (mydatabase), numeric characters (mydatabase2), dots (main.db), and dashes (enclosed within backticks, e.g., `CREATE DATABASE 'main-db'`).
- Names cannot end with dots or dashes.
- Names that begin with an underscore or with the prefix `system` are reserved for internal use.

When a database has been created, it will show up in the listing provided by the command `SHOW DATABASES`.

**Query**

```
SHOW DATABASES
```

Table 465. Result
25.2.1. Handling Existing Databases [Enterprise edition]

This command is optionally idempotent, with the default behavior to fail with an error if the database already exists. Appending `IF NOT EXISTS` to the command ensures that no error is returned and nothing happens should the database already exist. Adding `OR REPLACE` to the command will result in any existing database being deleted and a new one created.

Query

```
CREATE DATABASE customers IF NOT EXISTS
```

Query

```
CREATE OR REPLACE DATABASE customers
```

This is equivalent to running `DROP DATABASE customers IF EXISTS` followed by `CREATE DATABASE customers`.

ℹ️ The `IF NOT EXISTS` and `OR REPLACE` parts of this command cannot be used together.

25.2.2. Options [Enterprise edition]

The create database command can have a map of options, e.g. `OPTIONS { key : 'value'}`

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>existingData</td>
<td>use</td>
<td>Controls how the system handles existing data on disk when creating the database. Currently this is only supported with <code>existingDataSeedInstance</code> and must be set to <code>use</code> which indicates the existing data files should be used for the new database.</td>
</tr>
</tbody>
</table>
### ExistingData and ExistingDataSeedInstance

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>existingDataSeedInstance</td>
<td>instance ID of the cluster node</td>
<td>Defines which instance is used for seeding the data of the created database. The instance id can be taken from the id column of the <code>dbms.cluster.overview()</code> procedure. Can only be used in clusters.</td>
</tr>
</tbody>
</table>

> The `existingData` and `existingDataSeedInstance` options cannot be combined with the `OR REPLACE` part of this command.

### 25.3. Altering databases

Databases can be modified using the command `ALTER DATABASE`. For example, a database always has read-write access mode on creation, unless the configuration parameter `dbms.databases.default_to_read_only` is set to `true`. To change it to read-only, you can use the `ALTER DATABASE` command with the sub-clause `SET ACCESS READ ONLY`. Subsequently, the database access mode can be switched back to read-write using the sub-clause `SET ACCESS READ WRITE`. Altering the database access mode is allowed at all times, whether a database is online or offline.

Database access modes can also be managed using the configuration parameters `dbms.databases.default_to_read_only`, `dbms.databases.read_only`, and `dbms.database.writable`. For details, see [Configuration parameters](#). If conflicting modes are set by the `ALTER DATABASE` command and the configuration parameters, i.e. one says read-write and the other read-only, the database will be read-only and prevent write queries.

**Query**

```ruby
ALTER DATABASE customers SET ACCESS READ ONLY
```

0 rows, System updates: 1

The database access mode can be seen in the `access` output column of the command `SHOW DATABASES`.

**Query**

```ruby
SHOW DATABASES yield name, access
```

**Table 466. Result**

<table>
<thead>
<tr>
<th>name</th>
<th>access</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;customers&quot;</td>
<td>&quot;read-only&quot;</td>
</tr>
<tr>
<td>&quot;movies&quot;</td>
<td>&quot;read-write&quot;</td>
</tr>
<tr>
<td>&quot;neo4j&quot;</td>
<td>&quot;read-write&quot;</td>
</tr>
<tr>
<td>&quot;system&quot;</td>
<td>&quot;read-write&quot;</td>
</tr>
</tbody>
</table>

Rows: 4
This command is optionally idempotent, with the default behavior to fail with an error if the database does not exist. Appending `IF EXISTS` to the command ensures that no error is returned and nothing happens should the database not exist.

```
ALTER DATABASE nonExisting IF EXISTS SET ACCESS READ WRITE
```

### 25.4. Stopping databases **Enterprise edition**

Databases can be stopped using the command `STOP DATABASE`.

```
STOP DATABASE customers
```

0 rows, System updates: 1

The status of the stopped database can be seen using the command `SHOW DATABASE name`.

```
SHOW DATABASE customers
```

<table>
<thead>
<tr>
<th>name</th>
<th>aliases</th>
<th>access</th>
<th>address</th>
<th>role</th>
<th>requested Status</th>
<th>currentStatus</th>
<th>error</th>
<th>default</th>
<th>home</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;customers&quot;</td>
<td>[]</td>
<td>&quot;read-only&quot;</td>
<td>&quot;localhost:7687&quot;</td>
<td>&quot;standalone&quot;</td>
<td>&quot;offline&quot;</td>
<td>&quot;offline&quot;</td>
<td>&quot;&quot;</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

Rows: 1

### 25.5. Starting databases **Enterprise edition**

Databases can be started using the command `START DATABASE`.

```
START DATABASE customers
```

0 rows, System updates: 1

The status of the started database can be seen using the command `SHOW DATABASE name`.

```
SHOW DATABASE customers
```

Table 468. Result
25.6. Deleting databases

Databases can be deleted using the command `DROP DATABASE`.

Query

```sql
DROP DATABASE customers
```

0 rows, System updates: 1

When a database has been deleted, it will no longer show up in the listing provided by the command `SHOW DATABASES`.

Query

```sql
SHOW DATABASES
```

Table 469. Result

<table>
<thead>
<tr>
<th>name</th>
<th>aliases</th>
<th>access</th>
<th>address</th>
<th>role</th>
<th>requested Status</th>
<th>currentStatus</th>
<th>error</th>
<th>default</th>
<th>home</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;customers&quot;</td>
<td>[]</td>
<td>&quot;read-only&quot;</td>
<td>&quot;localhost:7687&quot;</td>
<td>&quot;standalone&quot;</td>
<td>&quot;online&quot;</td>
<td>&quot;online&quot;</td>
<td>&quot;&quot;</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

Rows: 1

This command is optionally idempotent, with the default behavior to fail with an error if the database does not exist. Appending `IF EXISTS` to the command ensures that no error is returned and nothing happens should the database not exist. It will always return an error, if there is an existing alias that targets the database. In that case, the alias needs to be dropped before dropping the database.

Query

```sql
DROP DATABASE customers IF EXISTS
```

The `DROP DATABASE` command will remove a database entirely. However, you can request that a dump of the store files is produced first, and stored in the path configured using the `dbmsdirectories.dumps.root` setting (by default `<neo4j-home>/data/dumps`). This can be achieved by appending `DUMP DATA` to the command (or `DESTROY DATA` to explicitly request the default behavior). These dumps are equivalent to those...
produced by `neo4j-admin dump` and can be similarly restored using `neo4j-admin load`.

Query

```sql
DROP DATABASE customers DUMP DATA
```

The options `IF EXISTS` and `DUMP DATA/DESTROY DATA` can also be combined. An example could look like this:

Query

```sql
DROP DATABASE customers IF EXISTS DUMP DATA
```

25.7. Wait options

Aside from `SHOW DATABASES` and `ALTER DATABASE`, all database management commands accept an optional `WAIT/NOWAIT` clause. The `WAIT/NOWAIT` clause allows you to specify a time limit in which the command must complete and return. The options are:

- `WAIT n SECONDS` - Return once completed or when the specified time limit of `n` seconds is up.
- `WAIT` - Return once completed or when the default time limit of 300 seconds is up.
- `NOWAIT` - Return immediately.

A command using a `WAIT` clause will automatically commit the current transaction when it executes successfully, as the command needs to run immediately for it to be possible to `WAIT` for it to complete. Any subsequent commands executed will therefore be performed in a new transaction. This is different to the usual transactional behavior, and for this reason it is recommended that these commands be run in their own transaction. The default behavior is `NOWAIT`, so if no clause is specified the transaction will behave normally and the action is performed in the background post-commit.

Query

```sql
CREATE DATABASE slow WAIT 5 SECONDS
```

Table 470. Result

<table>
<thead>
<tr>
<th>address</th>
<th>state</th>
<th>message</th>
<th>success</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;localhost:7687&quot;</td>
<td>&quot;CaughtUp&quot;</td>
<td>&quot;caught up&quot;</td>
<td>true</td>
</tr>
</tbody>
</table>

Rows: 1

The `success` column provides an aggregate status of whether or not the command is considered successful and thus every row will have the same value. The intention of this column is to make it easy to determine, for example in a script, whether or not the command completed successfully without timing out.

- A command with a `WAIT` clause may be interrupted whilst it is waiting to complete. In this event the command will continue to execute in the background and will not be aborted.
Chapter 26. Database alias management

This chapter explains how to use Cypher to manage database aliases in Neo4j.

There are two kinds of aliases, local database aliases and remote database aliases. A local database alias can only target a database within the same DBMS. A remote alias may target a database from another Neo4j DBMS. When a query is run against an alias, it will be redirected to the target database. The home database for users can be set to an alias, which will be resolved to the target database on use.

A local alias can be used in all other Cypher commands in place of the target database. Please note that the local alias will be resolved while executing the command. Privileges are defined on the database, and not the local alias.

A remote alias can be used for connecting to a database of a remote Neo4j DBMS, use clauses, setting a user’s home database and defining the access privileges to the remote database. Remote aliases require configuration to safely connect to the remote target, which is described in Connecting remote databases. It is not possible to impersonate a user on the remote database or to execute an administration command on the remote database via a remote alias.

Aliases can be created and managed using a set of Cypher administration commands executed against the system database. The required privileges are described here. When connected to the DBMS over bolt, administration commands are automatically routed to the system database.

The syntax of the alias management commands is as follows:

Table 471. Alias management command syntax

<table>
<thead>
<tr>
<th>Command</th>
<th>Syntax</th>
<th>Description</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Show Database Alias</td>
<td>SHOW ALIASES FOR DATABASE[S] [WHERE expression]</td>
<td>Lists both local and remote database aliases.</td>
<td>SHOW ALIASES FOR DATABASE[S] YIELD { *</td>
<td>field[, ...] } [ORDER BY field[, ...]] [SKIP n] [LIMIT n] [WHERE expression] [RETURN field[, ...] [ORDER BY field[, ...]] [SKIP n] [LIMIT n]]</td>
</tr>
<tr>
<td>Create Local Alias</td>
<td>CREATE ALIAS name [IF NOT EXISTS] FOR DATABASE targetName</td>
<td></td>
<td>CREATE OR REPLACE ALIAS name FOR DATABASE targetName</td>
<td></td>
</tr>
<tr>
<td>Command</td>
<td>Syntax</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>------------------------------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Create Remote Alias</td>
<td>CREATE ALIAS name [IF NOT EXISTS] FOR DATABASE targetName AT 'url' USER username PASSsword 'password' [DRIVER &quot;{&quot; setting: value[, ...] &quot;}&quot;]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CREATE OR REPLACE ALIAS name FOR DATABASE targetName AT 'url' USER username PASSsword 'password' [DRIVER &quot;{&quot; setting: value[, ...] &quot;}&quot;]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alter Local Alias</td>
<td>ALTER ALIAS name [IF EXISTS] SET DATABASE TARGET targetName</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alter Remote Alias</td>
<td>ALTER ALIAS name [IF EXISTS] SET DATABASE [TARGET targetName AT 'url'] [USER username] [PASSsword 'password'] [DRIVER &quot;{&quot; setting: value[, ...] &quot;}&quot;]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Drop Alias</td>
<td>DROP ALIAS name [IF EXISTS] FOR DATABASE</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Drop either a local or remote database alias.

This is the list of the allowed driver settings for remote aliases.

Table 472. ssl_enforced

<table>
<thead>
<tr>
<th>Description</th>
<th>SSL for remote alias drivers is configured through the target url scheme. If ssl_enforced is set to true, a secure url scheme is enforced. This will be validated when the command is executed.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid values</td>
<td>Boolean</td>
</tr>
<tr>
<td>Default value</td>
<td>true</td>
</tr>
</tbody>
</table>

Table 473. connection_timeout

<table>
<thead>
<tr>
<th>Description</th>
<th>Socket connection timeout. A timeout of zero is treated as an infinite timeout and will be bound by the timeout configured on the operating system level.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid values</td>
<td>Duration</td>
</tr>
<tr>
<td>Default value</td>
<td>dbms.routing.driver.connection.connect_timeout</td>
</tr>
</tbody>
</table>

Table 474. connection_max_lifetime
<table>
<thead>
<tr>
<th>Description</th>
<th>Pooled connections older than this threshold will be closed and removed from the pool. Setting this option to a low value will cause a high connection churn and might result in a performance hit. It is recommended to set maximum lifetime to a slightly smaller value than the one configured in network equipment (load balancer, proxy, firewall, etc. can also limit maximum connection lifetime).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid values</td>
<td>Duration. Zero and negative values result in lifetime not being checked.</td>
</tr>
<tr>
<td>Default value</td>
<td><code>dbms.routing.driver.connection.max_lifetime</code></td>
</tr>
</tbody>
</table>

**Table 475. connection_pool_acquisition_timeout**

<table>
<thead>
<tr>
<th>Description</th>
<th>Maximum amount of time spent attempting to acquire a connection from the connection pool. This timeout only kicks in when all existing connections are being used and no new connections can be created because maximum connection pool size has been reached. Error is raised when connection can’t be acquired within configured time.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid values</td>
<td>Duration. Negative values are allowed and result in unlimited acquisition timeout. Value of 0 is allowed and results in no timeout and immediate failure when connection is unavailable.</td>
</tr>
<tr>
<td>Default value</td>
<td><code>dbms.routing.driver.connection.pool.acquisition_timeout</code></td>
</tr>
</tbody>
</table>

**Table 476. connection_pool_max_size**

<table>
<thead>
<tr>
<th>Description</th>
<th>Maximum total number of connections to be managed by a connection pool. The limit is enforced for a combination of a host and user.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid values</td>
<td>Integer. Negative values are allowed and result in unlimited pool. Value of 0 is not allowed.</td>
</tr>
<tr>
<td>Default value</td>
<td><code>dbms.routing.driver.connection.pool.max_size</code></td>
</tr>
</tbody>
</table>

**Table 477. logging_level**

| Description | Sets level for driver internal logging. |
Valid values

org.neo4j.logging.Level.

One of DEBUG, INFO, WARN, ERROR, or NONE.

Default value
dbms.routing.driver.logging.level

If transaction modifies an alias, other transactions concurrently executing against that alias may be aborted and rolled back for safety. This prevents issues such as a transaction executing against multiple target databases for the same alias.

26.1. Listing database aliases

Available database aliases can be seen using `SHOW ALIASES FOR DATABASE`. The required privileges are described here. `SHOW ALIASES FOR DATABASE` will produce a table of database aliases with the following columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Default Output</th>
<th>Full Output</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name of the database alias.</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>database</td>
<td>The names of the target database.</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>location</td>
<td>The location of the database, either local or remote.</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>url</td>
<td>Target location or null if the target is local.</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>user</td>
<td>User connecting to the remote database or null if the target database is local.</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>driver</td>
<td>The driver options for connection to the remote database or null if the target database is local or if no driver settings are added.</td>
<td>✔️</td>
<td>✔️</td>
<td>List of driver settings allowed for remote database aliases.</td>
</tr>
</tbody>
</table>

A summary of all available databases alias can be displayed using the command `SHOW ALIASES FOR DATABASE`.

Query

SHOW ALIASES FOR DATABASE

Table 478. Result

<table>
<thead>
<tr>
<th>name</th>
<th>database</th>
<th>location</th>
<th>url</th>
<th>user</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;films&quot;</td>
<td>&quot;movies&quot;</td>
<td>&quot;local&quot;</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;motion pictures&quot;</td>
<td>&quot;movies&quot;</td>
<td>&quot;local&quot;</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;movie scripts&quot;</td>
<td>&quot;scripts&quot;</td>
<td>&quot;remote&quot;</td>
<td>&quot;neo4j+s://location:7687&quot;</td>
<td>&quot;alice&quot;</td>
</tr>
</tbody>
</table>

Rows: 3
The detailed information for a particular database alias can be displayed using the command `SHOW ALIASES FOR DATABASE YIELD *`. When a `YIELD *` clause is provided, the full set of columns is returned.

Query

```
SHOW ALIASES FOR DATABASE YIELD *
```

Table 479. Result

<table>
<thead>
<tr>
<th>name</th>
<th>database</th>
<th>location</th>
<th>url</th>
<th>user</th>
<th>driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;films&quot;</td>
<td>&quot;movies&quot;</td>
<td>&quot;local&quot;</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;motion pictures&quot;</td>
<td>&quot;movies&quot;</td>
<td>&quot;local&quot;</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;movie scripts&quot;</td>
<td>&quot;scripts&quot;</td>
<td>&quot;remote&quot;</td>
<td>&quot;neo4j+s://location:7687&quot;</td>
<td>&quot;alice&quot;</td>
<td>{connection_pool_max_size -&gt; 10, connection_pool_idle_test -&gt; PT2M, connection_pool_acquisition_timeout -&gt; PT1M, connection_max_lifetime -&gt; PT1H, logging_level -&gt; &quot;INFO&quot;, ssl_enforced -&gt; true, connection_timeout -&gt; PT5S}</td>
</tr>
</tbody>
</table>

Rows: 3

The number of database aliases can be seen using a `count()` aggregation with `YIELD` and `RETURN`.

Query

```
SHOW ALIASES FOR DATABASE YIELD * RETURN count(*) as count
```

Table 480. Result

<table>
<thead>
<tr>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

Rows: 1

It is also possible to filter and sort the results by using `YIELD`, `ORDER BY` and `WHERE`.

Query

```
SHOW ALIASES FOR DATABASE YIELD name, url, database ORDER BY database WHERE name CONTAINS 'e'
```

In this example:

- The number of columns returned has been reduced with the `YIELD` clause.
- The order of the returned columns has been changed.
- The results have been filtered to only show database alias names containing 'e'.

444
The results are ordered by the 'database' column using ORDER BY.

It is also possible to use SKIP and LIMIT to paginate the results.

Table 481. Result

<table>
<thead>
<tr>
<th>name</th>
<th>url</th>
<th>database</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;motion pictures&quot;</td>
<td>&lt;null&gt;</td>
<td>&quot;movies&quot;</td>
</tr>
<tr>
<td>&quot;movie scripts&quot;</td>
<td>&quot;neo4j+s://location:7687&quot;</td>
<td>&quot;scripts&quot;</td>
</tr>
</tbody>
</table>

Rows: 2

26.2. Creating database aliases

Aliases can be created using CREATE ALIAS. The required privileges are described here.

This command is optionally idempotent, with the default behavior to fail with an error if the database alias already exists. Inserting IF NOT EXISTS after the alias name ensures that no error is returned and nothing happens should a database alias with that name already exist. Adding OR REPLACE to the command will result in any existing database alias being deleted and a new one created. CREATE OR REPLACE ALIAS will fail if there is an existing database with the same name.

The IF NOT EXISTS and OR REPLACE parts of this command cannot be used together.

Alias names are subject to the standard Cypher restrictions on valid identifiers. The following naming rules apply:

- A name is a valid identifier, additionally allowing dots e.g. main.alias for local aliases.
- Name length can be up to 65534 characters.
- Names cannot end with dots.
- Names that begin with an underscore or with the prefix system are reserved for internal use.
- Non-alphabetic characters, including numbers, symbols and whitespace characters, can be used in names, but must be escaped using backticks.

26.2.1. Creating local database aliases

Local aliases are created with a target database.

Query

```cypher
CREATE ALIAS 'northwind' FOR DATABASE 'northwind-graph-2021'
```

0 rows, System updates: 1

When a local database alias has been created, it will show up in the aliases column provided by the
command `SHOW DATABASES` and in the `SHOW ALIASES FOR DATABASE` command.

Query

```
SHOW DATABASE 'northwind'
```

Table 482. Result

<table>
<thead>
<tr>
<th>name</th>
<th>aliases</th>
<th>access</th>
<th>address</th>
<th>role</th>
<th>requested Status</th>
<th>currentStatus</th>
<th>error</th>
<th>default</th>
<th>home</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;northwind-graph-2021&quot;</td>
<td>[&quot;northwind&quot;] &quot;read-write&quot;</td>
<td>localhost:7687</td>
<td>&quot;standalone&quot;</td>
<td>&quot;online&quot;</td>
<td>&quot;online&quot;</td>
<td>&quot;&quot;</td>
<td>false</td>
<td>false</td>
<td></td>
</tr>
</tbody>
</table>

Rows: 1

Query

```
SHOW ALIASES FOR DATABASE WHERE name = 'northwind'
```

Table 483. Result

<table>
<thead>
<tr>
<th>name</th>
<th>database</th>
<th>location</th>
<th>url</th>
<th>user</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;northwind&quot;</td>
<td>&quot;northwind-graph-2021&quot;</td>
<td>&quot;local&quot;</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

Rows: 1

Adding a local alias with the same name as an existing local or remote alias will do nothing with the `IF NOT EXISTS` clause but fail without it.

Query

```
CREATE ALIAS 'northwind' IF NOT EXISTS FOR DATABASE 'northwind-graph-2020'
```

0 rows

It is also possible to replace an alias. The old alias may be either local or remote.

Query

```
CREATE OR REPLACE ALIAS 'northwind' FOR DATABASE 'northwind-graph-2020'
```

0 rows, System updates: 2

This is equivalent to running `DROP ALIAS northwind IF EXISTS FOR DATABASE` followed by `CREATE ALIAS northwind FOR DATABASE northwind-graph-2020`.

26.2.2. Creating remote database aliases

Database aliases can also point to remote databases by providing an url and the credentials of a user on the remote Neo4j DBMS. See Connecting remote databases for the necessary configurations.
It is possible to override the default driver settings per alias, which are used for connecting to the remote database. The full list of supported driver settings can be seen [here](#).

When a database alias pointing to a remote database has been created, its details can be shown with the `SHOW ALIASES FOR DATABASE` command.

<table>
<thead>
<tr>
<th>name</th>
<th>database</th>
<th>location</th>
<th>url</th>
<th>user</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;remote-northwind&quot;</td>
<td>&quot;northwind-graph-2020&quot;</td>
<td>&quot;remote&quot;</td>
<td>&quot;neo4j+s://location:7687&quot;</td>
<td>&quot;alice&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

Creating remote aliases also allows `IF NOT EXISTS` and `OR REPLACE` clauses. Both check for any remote or local database aliases.
26.3. Altering database aliases

Aliases can be altered using `ALTER ALIAS` to change its database target, url, user credentials, or driver settings. The required privileges are described [here](#). Only the clauses used will be altered.

Local aliases can not be altered to remote aliases or vice versa.

Example of altering a local alias target.

Query

```sql
ALTER ALIAS 'northwind' SET DATABASE TARGET 'northwind-graph-2021'
```

0 rows, System updates: 1

Example of altering a remote alias target.

Query

```sql
ALTER ALIAS 'remote-northwind' SET DATABASE TARGET 'northwind-graph-2020' AT "neo4j+s://other-location:7687"
```

0 rows, System updates: 1

Example of altering a remote alias credentials and driver settings.

Query

```sql
ALTER ALIAS 'remote-with-driver-settings' SET DATABASE USER bob PASSWORD 'newPassword'
  DRIVER {
    connection_timeout: duration({ minutes: 1 }),
    logging_level: "debug"
  }
```

0 rows, System updates: 1

All driver settings are replaced by the new ones. In this case, by not repeating the driver setting `connection_pool_max_size` the value will be deleted and fallback to the default value.

Example of altering a remote alias to remove all custom driver settings.

Query

```sql
ALTER ALIAS 'movie scripts' SET DATABASE DRIVER {
}
```

0 rows, System updates: 1

When a local database alias has been altered, it will show up in the aliases column for the target database provided by the command `SHOW DATABASES`.

Query

```sql
SHOW DATABASE 'northwind'
```
Table 486. Result

<table>
<thead>
<tr>
<th>name</th>
<th>aliases</th>
<th>access</th>
<th>address</th>
<th>role</th>
<th>requested Status</th>
<th>currentStatus</th>
<th>error</th>
<th>default</th>
<th>home</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;northwind-graph-2021&quot;</td>
<td>[&quot;northwind&quot;]</td>
<td>&quot;read-write&quot;</td>
<td>&quot;localhost:7687&quot;</td>
<td>&quot;standalone&quot;</td>
<td>&quot;online&quot;</td>
<td>&quot;online&quot;</td>
<td>&quot;&quot;</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

.Rows: 1

The changes for all database aliases will show up in the SHOW ALIASES FOR DATABASE command.

Query

```
SHOW ALIASES FOR DATABASE YIELD * WHERE name in ['northwind', 'remote-northwind', 'remote-with-driver-settings', 'movie scripts']
```

Table 487. Result

<table>
<thead>
<tr>
<th>name</th>
<th>database</th>
<th>location</th>
<th>url</th>
<th>user</th>
<th>driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;movie scripts&quot;</td>
<td>&quot;scripts&quot;</td>
<td>&quot;remote&quot;</td>
<td>&quot;neo4j+s://location:7687&quot;</td>
<td>&quot;alice&quot;</td>
<td>()</td>
</tr>
<tr>
<td>&quot;northwind&quot;</td>
<td>&quot;northwind-graph-2021&quot;</td>
<td>&quot;local&quot;</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;remote-northwind&quot;</td>
<td>&quot;northwind-graph-2020&quot;</td>
<td>&quot;remote&quot;</td>
<td>&quot;neo4j+s://other-location:7687&quot;</td>
<td>&quot;alice&quot;</td>
<td>{}</td>
</tr>
<tr>
<td>&quot;remote-with-driver-settings&quot;</td>
<td>&quot;northwind-graph-2020&quot;</td>
<td>&quot;remote&quot;</td>
<td>&quot;neo4j+s://location:7687&quot;</td>
<td>&quot;bob&quot;</td>
<td>{logging_level -&gt; &quot;DEBUG&quot;, connection_timeout -&gt; PT1M}</td>
</tr>
</tbody>
</table>

.Rows: 4

This command is optionally idempotent, with the default behavior to fail with an error if the alias does not exist. Appending IF EXISTS to the command ensures that no error is returned and nothing happens should the alias not exist.

Query

```
ALTER ALIAS 'no-alias' IF EXISTS SET DATABASE TARGET 'northwind-graph-2021'
```

0 rows

26.4. Deleting database aliases

Both local and remote aliases can be deleted using the DROP ALIAS command. The required privileges are described here.

Query

```
DROP ALIAS 'northwind' FOR DATABASE
```

0 rows, System updates: 1
When a database alias has been deleted, it will no longer show up in the aliases column provided by the command `SHOW DATABASES` or in the `SHOW ALIASES FOR DATABASE` command.

This command is optionally idempotent, with the default behavior to fail with an error if the alias does not exist. Inserting `IF EXISTS` after the alias name ensures that no error is returned and nothing happens should the alias not exist.
Chapter 27. Access control

This chapter explains how to manage Neo4j role-based access control and fine-grained security.

Neo4j has a complex security model stored in the system graph, which is maintained on a special database called the system database. All administrative commands need to be executed against the system database. When connected to the DBMS over bolt, administrative commands are automatically routed to the system database. For more information on how to manage multiple databases, refer to the section on administering databases.

The concept of role-based access control was introduced in Neo4j 3.1. Since then, it has been possible to create users and assign them to roles to control whether users can read, write and administer the database. In Neo4j 4.0 this model was enhanced significantly with the addition of privileges, which are the underlying access-control rules by which the users rights are defined.

The original built-in roles still exist with almost the exact same access rights, but they are no-longer statically defined (see Built-in roles). Instead they are defined in terms of their underlying privileges and they can be modified by adding an removing these access rights.

In addition, any new roles can be assigned to any combination of privileges to create specific access controls. Another new major capability is the sub-graph access control, through which read access to the graph can be limited to specific combinations of labels, relationship types, and properties.

27.1. Syntax summaries

Almost all administration commands have variations. The most common are parts of the command that are optional or that can have multiple values.

See below a summary of the syntax to check all versions of a command. These summaries use some special characters to indicate such variations.

<table>
<thead>
<tr>
<th>Special characters in syntax summaries</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Or. Used to indicate alternative parts of a command. Needs to be part of a grouping.</td>
</tr>
<tr>
<td></td>
<td>If the syntax needs to specify either a name or *, this can be indicated with *</td>
</tr>
<tr>
<td>{ and }</td>
<td>Used to group parts of the command. Commonly found together with</td>
</tr>
<tr>
<td></td>
<td>In order to use the or in the syntax summary, it needs to be in a group: {*</td>
</tr>
<tr>
<td>Character</td>
<td>Meaning</td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
</tr>
<tr>
<td>[ and ]</td>
<td>Used to indicate an optional part of the command. It also groups alternatives together, when there can be either of the alternatives or nothing.</td>
</tr>
<tr>
<td>...</td>
<td>Repeated pattern. Related to the command part immediately before this is repeated.</td>
</tr>
</tbody>
</table>
| "         | When a special character is part of the syntax itself, we surround it with " to indicate this. | To include `{` in the syntax use 

```
"{ { * | name } "
```

In this case, you will get either `{*}` or `{name}`. |

The special characters in the table above are the only ones that need to be escaped using " in the syntax summaries.

Here is an example that uses all special characters is granting the `READ` privilege:

```cypher
GRANT READ
"{ { * | property[, ...] } "
ON {HOME | GRAPH | GRAPH[S] { * | name[, ...] }}
[
    ELEMENT[S] { * | label-or-rel-type[, ...] }
| NODE[S] { * | label[, ...] }
| RELATIONSHIP[S] { * | rel-type[, ...] }
] TO role[, ...]
```

Note that this command includes ( and ) in the syntax, and between them there can be a grouping of properties or the character *. It also has multiple optional parts, including the entity part of the command which is the grouping following the graph name.

However, there is no need to escape any characters when creating a constraint for a node property. This is because ( and ) are not special characters, and [ and ] indicate that the constraint name is optional, and therefore not part of the command.

```cypher
CREATE CONSTRAINT [constraint_name] [IF NOT EXISTS]
FOR (n:LabelName)
REQUIRE n.propertyName IS NOT NULL
```

### 27.2. Managing users

This section explains how to use Cypher to manage users in Neo4j.

Users can be created and managed using a set of Cypher administration commands executed against the system database. When connected to the DBMS over bolt, administration commands are automatically routed to the system database.
# 27.2.1. User management command syntax

<table>
<thead>
<tr>
<th>Command</th>
<th>SHOW CURRENT USER</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td><code>SHOW CURRENT USER</code> &lt;br/&gt; `[YIELD { *</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Lists the current user. &lt;br/&gt; When using the <code>RETURN</code> clause, the <code>YIELD</code> clause is mandatory and must not be omitted. &lt;br/&gt; For more information, see Listing current user.</td>
</tr>
<tr>
<td><strong>Required privilege</strong></td>
<td>None</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command</th>
<th>SHOW USERS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td><code>SHOW USERS</code> &lt;br/&gt; `[YIELD { *</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Lists all users. &lt;br/&gt; When using the <code>RETURN</code> clause, the <code>YIELD</code> clause is mandatory and must not be omitted. &lt;br/&gt; For more information, see Listing users.</td>
</tr>
<tr>
<td><strong>Required privilege</strong></td>
<td>GRANT SHOW USER &lt;br/&gt; (see DBMS USER MANAGEMENT privileges)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command</th>
<th>SHOW USER PRIVILEGES</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td><code>SHOW USER[S] [name[, ...]] PRIVILEGE[S] [AS [REVOKE] COMMAND[S]]</code> &lt;br/&gt; `[YIELD { *</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Lists the privileges granted to the specified users or the current user if no user is specified. &lt;br/&gt; When using the <code>RETURN</code> clause, the <code>YIELD</code> clause is mandatory and must not be omitted. &lt;br/&gt; For more information, see Listing privileges.</td>
</tr>
<tr>
<td>Command</td>
<td>CREATE USER</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Syntax</td>
<td>CREATE USER name [IF NOT EXISTS] SET [PLAINTEXT</td>
</tr>
<tr>
<td>Description</td>
<td>Creates a new user. For more information, see Creating users.</td>
</tr>
<tr>
<td>Required privilege</td>
<td>GRANT CREATE USER</td>
</tr>
<tr>
<td></td>
<td>(see DBMS USER MANAGEMENT privileges)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command</th>
<th>CREATE OR REPLACE USER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>CREATE OR REPLACE USER name SET [PLAINTEXT</td>
</tr>
<tr>
<td>Description</td>
<td>Creates a new user, or if a user with the same name exists, replace it. For more information, see Creating users.</td>
</tr>
<tr>
<td>Required privilege</td>
<td>GRANT CREATE USER and GRANT DROP USER</td>
</tr>
<tr>
<td></td>
<td>(see DBMS USER MANAGEMENT privileges)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command</th>
<th>RENAME USER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>RENAME USER name [IF EXISTS] TO otherName</td>
</tr>
</tbody>
</table>

Required privilege

GRANT SHOW PRIVILEGE

(see DBMS PRIVILEGE MANAGEMENT privileges)

GRANT SHOW USER

(see DBMS USER MANAGEMENT privileges)
| Description | Changes the name of a user.  
For more information, see [Renaming users](#). |
|---|---|
| Required privilege | GRANT RENAME USER  
(see [DBMS USER MANAGEMENT privileges](#)) |
| Command | **ALTER USER** |
| Syntax | ```sql
ALTER USER name [IF EXISTS]  
[SET [PLAINTEXT | ENCRYPTED] PASSWORD 'password']  
[[SET PASSWORD] CHANGE [NOT] REQUIRED]  
[SET STATUS {ACTIVE | SUSPENDED}]  
[SET HOME DATABASE name]  
[REMOVE HOME DATABASE]
``` |
| Description | Modifies the settings for an existing user. At least one `SET` or `REMOVE` clause is required.  
`SET` and `REMOVE` clauses cannot be combined in the same command.  
For more information, see [Modifying users](#). |
| Required privilege | GRANT SET PASSWORD, GRANT SET USER STATUS, and/or GRANT SET USER HOME DATABASE  
(see [DBMS USER MANAGEMENT privileges](#)) |
| Command | **ALTER CURRENT USER SET PASSWORD** |
| Syntax | ```sql
ALTER CURRENT USER SET PASSWORD FROM 'oldPassword' TO 'newPassword'
``` |
| Description | Change the current user’s password.  
For more information, see [Changing the current user’s password](#). |
| Required privilege | None |
| Command | **DROP USER** |
| Syntax | ```sql
DROP USER name [IF EXISTS]
``` |
| Description | Removes an existing user.  
For more information, see [Delete users](#). |
27.2.2. Listing current user

The currently logged-in user can be seen using `SHOW CURRENT USER`, which will produce a table with the following columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Community Edition</th>
<th>Enterprise Edition</th>
</tr>
</thead>
<tbody>
<tr>
<td>user</td>
<td>User name</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>roles</td>
<td>Roles granted to the user.</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>passwordChangeRequired</td>
<td>If true, the user must change their password at the next login.</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>suspended</td>
<td>If true, the user is currently suspended (cannot log in).</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>home</td>
<td>The home database configured by the user, or null if no home database has been configured. If this database is unavailable and the user does not specify a database to use, they will not be able to log in.</td>
<td>×</td>
<td>✓</td>
</tr>
</tbody>
</table>

SHOW CURRENT USER

Table 491. Result

<table>
<thead>
<tr>
<th>user</th>
<th>roles</th>
<th>passwordChangeRequired</th>
<th>suspended</th>
<th>home</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>jake</em></td>
<td>[*&quot;PUBLIC&quot;]</td>
<td>false</td>
<td>false</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

Rows: 1

This command is only supported for a logged-in user and will return an empty result if authorization has been disabled.

27.2.3. Listing users

Available users can be seen using `SHOW USERS`, which will produce a table of users with the following columns:
<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Community Edition</th>
<th>Enterprise Edition</th>
</tr>
</thead>
<tbody>
<tr>
<td>user</td>
<td>User name</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>roles</td>
<td>Roles granted to the user.</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>passwordChangeRequired</td>
<td>If true, the user must change their password at the next login.</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>suspended</td>
<td>If true, the user is currently suspended (cannot log in).</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>home</td>
<td>The home database configured by the user, or null if no home database has been configured.</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Table 492. Result**

<table>
<thead>
<tr>
<th>user</th>
<th>roles</th>
<th>passwordChangeRequired</th>
<th>suspended</th>
<th>home</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;neo4j&quot;</td>
<td>[&quot;admin&quot;,&quot;PUBLIC&quot;]</td>
<td>true</td>
<td>false</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

Rows: 1

When first starting a Neo4j DBMS, there is always a single default user neo4j with administrative privileges. It is possible to set the initial password using `neo4j-admin set-initial-password`, otherwise it is necessary to change the password after the first login.

Example 6. Show user

This example shows how to:

- Reorder the columns using a `YIELD` clause.
- Filter the results using a `WHERE` clause.

```
SHOW USERS YIELD user, suspended, passwordChangeRequired, roles, home
WHERE user = 'jake'
```
Example 7. Show user

It is possible to add a \texttt{RETURN} clause to further manipulate the results after filtering. In this example, the \texttt{RETURN} clause is used to filter out the \texttt{roles} column and rename the \texttt{user} column to \texttt{adminUser}.

\begin{verbatim}
SHOW USERS YIELD roles, user
WHERE 'admin' IN roles
RETURN user AS adminUser
\end{verbatim}

The \texttt{SHOW USER name PRIVILEGES} command is described in Listing privileges.

27.2.4. Creating users

Users can be created using \texttt{CREATE USER}.

\begin{verbatim}
CREATE USER name [IF NOT EXISTS]
SET [PLAINTEXT | ENCRYPTED] PASSWORD 'password'
[[SET PASSWORD] CHANGE [NOT] REQUIRED]
[SET STATUS {ACTIVE | SUSPENDED}]
[SET HOME DATABASE name]
\end{verbatim}

Users can be created or replaced using \texttt{CREATE OR REPLACE USER}.

\begin{verbatim}
CREATE OR REPLACE USER name
SET [PLAINTEXT | ENCRYPTED] PASSWORD 'password'
[[SET PASSWORD] CHANGE [NOT] REQUIRED]
[SET STATUS {ACTIVE | SUSPENDED}]
[SET HOME DATABASE name]
\end{verbatim}

- For \texttt{SET PASSWORD}:
  - The \texttt{password} can either be a string value or a string parameter.
  - All passwords are encrypted (hashed) when stored in the Neo4j system database. \texttt{PLAINTEXT} and \texttt{ENCRYPTED} just refer to the format of the password in the Cypher command, i.e. whether Neo4j needs to hash it or it has already been hashed. Consequently, it is never possible to get the plaintext of a password back out of the database. A password can be set in either fashion at any time.
  - The optional \texttt{PLAINTEXT} in \texttt{SET PLAINTEXT PASSWORD} has the same behavior as \texttt{SET PASSWORD}.
  - The optional \texttt{ENCRYPTED} is used to recreate an existing user when the plaintext password is unknown, but the encrypted password is available in the data/scripts/databasename/restore_metadata.cypher file of a database backup. See Operations Manual $\rightarrow$ Restore a database backup $\rightarrow$ Example.

With \texttt{ENCRYPTED}, the password string is expected to be in the format of \texttt{<encryption-version>,<hash> or <salt>}, where, for example:

- 0 is the first version and refers to the SHA-256 cryptographic hash function with iterations 1.
- 1 is the second version and refers to the SHA-256 cryptographic hash function with iterations 1024.
• If the optional SET PASSWORD CHANGE [NOT] REQUIRED is omitted, the default is CHANGE REQUIRED. The SET PASSWORD part is only optional if it directly follows the SET PASSWORD clause.

• The default for SET STATUS is ACTIVE.

• SET HOME DATABASE can be used to configure a home database for a user. A home database will be resolved if it is either pointing to a database or a database alias. If no home database is set, the DBMS default database is used as the home database for the user.

• The SET PASSWORD CHANGE [NOT] REQUIRED, SET STATUS, and SET HOME DATABASE clauses can be applied in any order.

Informational note

User names are case sensitive. The created user will appear on the list provided by SHOW USERS.

• In Neo4j Community Edition there are no roles, but all users have implied administrator privileges.

• In Neo4j Enterprise Edition all users are automatically assigned the PUBLIC role, giving them a base set of privileges.

Example 8. Create user

For example, you can create the user jake in a suspended state, with the home database anotherDb, and the requirement to change the password by using the command:

```sql
CREATE USER jake
SET PASSWORD 'abc'
CHANGE REQUIRED
SET STATUS SUSPENDED
SET HOME DATABASE anotherDb
```

Example 9. Create user

Or you can recreate the user jake in an active state, with an encrypted password (taken from the data/scripts/databasename/restore_metadata.cypher of a database backup), and the requirement to not change the password by running:

```sql
CREATE USER jake
SET ENCRYPTED PASSWORD '1,6d57a5e0b3317055454e455f96c98c750c77fb371f3f0634a1b8ff2a55c5b825,190ae47c661e0668a0c8be8a21ff78a4a34cfd918cae3c407e907b73932bd16c'
CHANGE NOT REQUIRED
SET STATUS ACTIVE
```

Informational note

The SET STATUS (ACTIVE | SUSPENDED) and SET HOME DATABASE parts of the commands are only available in Neo4j Enterprise Edition.

The CREATE USER command is optionally idempotent, with the default behavior to throw an exception if the user already exists. Appending IF NOT EXISTS to the CREATE USER command will ensure that no exception is thrown and nothing happens should the user already exist.
Example 10. Create user if not exists

```
CREATE USER jake IF NOT EXISTS
SET PLAINTEXT PASSWORD 'xyz'
```

The `CREATE OR REPLACE USER` command will result in any existing user being deleted and a new one created.

Example 11. Create or replace user

```
CREATE OR REPLACE USER jake
SET PLAINTEXT PASSWORD 'xyz'
```

This is equivalent to running `DROP USER jake IF EXISTS` followed by `CREATE USER jake SET PASSWORD 'xyz'`.

The `CREATE OR REPLACE USER` command does not allow the use of `IF NOT EXISTS`.

27.2.5. Renaming users

Users can be renamed with the `RENAME USER` command.

```
RENAME USER jake TO bob
```

```
SHOW USERS
```

Table 493. Result

<table>
<thead>
<tr>
<th>user</th>
<th>roles</th>
<th>passwordChangeRequired</th>
<th>suspended</th>
<th>home</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;bob&quot;</td>
<td>[&quot;PUBLIC&quot;]</td>
<td>true</td>
<td>false</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;neo4j&quot;</td>
<td>[&quot;admin&quot;,&quot;PUBLIC&quot;]</td>
<td>true</td>
<td>false</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

Rows: 2

The `RENAME USER` command is only available when using native authentication and authorization.

27.2.6. Modifying users

Users can be modified with `ALTER USER`.

460
ALTER USER name [IF EXISTS]
[SET [PLAINTEXT | ENCRYPTED] PASSWORD 'password']
[[SET PASSWORD] CHANGE [NOT] REQUIRED]
[SET STATUS {ACTIVE | SUSPENDED}]
[SET HOME DATABASE name]
[REMOVE HOME DATABASE name]

- At least one SET or REMOVE clause is required for the command.
- SET and REMOVE clauses cannot be combined in the same command.
- The SET PASSWORD CHANGE [NOT] REQUIRED, SET STATUS, and SET HOME DATABASE clauses can be applied in any order. The SET PASSWORD clause must come first, if used.
- For SET PASSWORD:
  - The password can either be a string value or a string parameter.
  - All passwords are encrypted (hashed) when stored in the Neo4j system database. PLAINTEXT and ENCRYPTED just refer to the format of the password in the Cypher command, i.e. whether Neo4j needs to hash it or it has already been hashed. Consequently, it is never possible to get the plaintext of a password back out of the database. A password can be set in either fashion at any time.
  - The optional PLAINTEXT in SET PLAINTEXT PASSWORD has the same behavior as SET PASSWORD.
  - The optional ENCRYPTED is used to update an existing user’s password when the plaintext password is unknown, but the encrypted password is available in the data/scripts/databasename/restore_metadata.cypher file of a database backup. See Operations Manual → Restore a database backup → Example.
    With ENCRYPTED, the password string is expected to be in the format of `<encryption-version>,<hash>` or `'<salt>', where, for example:
    - 0 is the first version and refers to the SHA-256 cryptographic hash function with iterations 1.
    - 1 is the second version and refers to the SHA-256 cryptographic hash function with iterations 1024.
  - If the optional SET PASSWORD CHANGE [NOT] REQUIRED is omitted, the default is CHANGE REQUIRED. The SET PASSWORD part is only optional if it directly follows the SET PASSWORD clause.
- For SET PASSWORD CHANGE [NOT] REQUIRED, the SET PASSWORD is only optional if it directly follows the SET PASSWORD clause.
- SET HOME DATABASE can be used to configure a home database for a user. A home database will be resolved if it is either pointing to a database or a database alias. If no home database is set, the DBMS default database is used as the home database for the user.
- REMOVE HOME DATABASE is used to unset the home database for a user. This results in the DBMS default database being used as the home database for the user.

For example, you can modify the user bob with a new password and active status, and remove the requirement to change his password:

```
ALTER USER bob SET PASSWORD 'abc123' CHANGE NOT REQUIRED SET STATUS ACTIVE
```

Or you may decide to assign the user bob a different home database:
ALTER USER bob SET HOME DATABASE anotherDbOrAlias

Or remove the home database from the user bob:

ALTER USER bob REMOVE HOME DATABASE

When altering a user, it is only necessary to specify the changes required. For example, leaving out the `CHANGE [NOT] REQUIRED` part of the query will leave that unchanged.

The `SET STATUS {ACTIVE | SUSPENDED}`, `SET HOME DATABASE`, and `REMOVE HOME DATABASE` parts of the command are only available in Neo4j Enterprise Edition.

The changes to the user will appear on the list provided by `SHOW USERS`:

```
SHOW USERS
```

<table>
<thead>
<tr>
<th>user</th>
<th>roles</th>
<th>passwordChangeRequired</th>
<th>suspended</th>
<th>home</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;bob&quot;</td>
<td>[&quot;PUBLIC&quot;]</td>
<td>false</td>
<td>false</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;neo4j&quot;</td>
<td>[&quot;admin&quot;,&quot;PUBLIC&quot;]</td>
<td>true</td>
<td>false</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

Rows: 2

The default behavior of this command is to throw an exception if the user does not exist. Adding an optional parameter `IF EXISTS` to the command makes it idempotent and ensures that no exception is thrown. Nothing happens should the user not exist.

```
ALTER USER nonExistingUser IF EXISTS SET PASSWORD 'abc123'
```

27.2.7. Changing the current user’s password

Users can change their password using `ALTER CURRENT USER SET PASSWORD`. The old password is required in addition to the new one, and either or both can be a string value or a string parameter. When a user executes this command it will change their password as well as set the `CHANGE NOT REQUIRED` flag.

```
ALTER CURRENT USER SET PASSWORD FROM 'abc123' TO '123xyz'
```

This command works only for a logged-in user and cannot be run with auth disabled.

27.2.8. Delete users

Users can be deleted with `DROP USER`. 
Deleting a user will not automatically terminate associated connections, sessions, transactions, or queries. However, when a user has been deleted, it will no longer appear on the list provided by `SHOW USERS`:

```
SHOW USERS
```

<table>
<thead>
<tr>
<th>user</th>
<th>roles</th>
<th>passwordChangeRequired</th>
<th>suspended</th>
<th>home</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;neo4j&quot;</td>
<td>[&quot;admin&quot;,&quot;PUBLIC&quot;]</td>
<td>true</td>
<td>false</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

Rows: 1

### 27.3. Managing roles

**Enterprise edition**

This section explains how to use Cypher to manage roles in Neo4j.

Roles can be created and managed using a set of Cypher administration commands executed against the system database.

When connected to the DBMS over bolt, administration commands are automatically routed to the system database.

#### 27.3.1. Role management command syntax

**Command**

```
SHOW ROLES
```

**Syntax**

```
SHOW [ALL|POPULATED] ROLES
  [YIELD { * | field[, ..] } [ORDER BY field[, ..]] [SKIP n] [LIMIT n]]
  [WHERE expression]
  [RETURN field[, ..] [ORDER BY field[, ..]] [SKIP n] [LIMIT n]]
```

**Description**

Lists roles.

When using the `RETURN` clause, the `YIELD` clause is mandatory and must not be omitted.

For more information, see Listing roles.

**Required privilege**

```
GRANT SHOW ROLE
```

(see DBMS ROLE MANAGEMENT privileges)

**Command**

```
SHOW ROLES WITH USERS
```
Syntax

SHOW [ALL | POPULATED] ROLES WITH USERS
[YIELD { * | field[, ...] } [ORDER BY field[, ...]] [SKIP n] [LIMIT n]]
[WHERE expression]
[RETURN field[, ...] [ORDER BY field[, ...]] [SKIP n] [LIMIT n]]

Description

Lists roles and users assigned to them.

When using the RETURN clause, the YIELD clause is mandatory and must not be omitted.

For more information, see Listing roles.

Required privilege

GRANT SHOW ROLE

(see DBMS ROLE MANAGEMENT privileges)

GRANT SHOW USER

(see DBMS USER MANAGEMENT privileges)

Command

SHOW ROLE PRIVILEGES

Syntax

SHOW ROLE[S] name[, ...] PRIVILEGE[S] [AS [REVOKE] COMMAND[S]]
[YIELD { * | field[, ...] } [ORDER BY field[, ...]] [SKIP n] [LIMIT n]]
[WHERE expression]
[RETURN field[, ...] [ORDER BY field[, ...]] [SKIP n] [LIMIT n]]

Description

Lists the privileges granted to the specified roles.

When using the RETURN clause, the YIELD clause is mandatory and must not be omitted.

For more information, see Listing privileges.

Required privilege

GRANT SHOW PRIVILEGE

(see DBMS PRIVILEGE MANAGEMENT privileges)

Command

CREATE ROLE

Syntax

CREATE ROLE name [IF NOT EXISTS] [AS COPY OF otherName]

Description

Creates a new role.

For more information, see Creating roles.
<table>
<thead>
<tr>
<th>Command</th>
<th>Syntax</th>
<th>Description</th>
<th>Required privilege</th>
<th>(see DBMS ROLE MANAGEMENT privileges)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE OR REPLACE ROLE</td>
<td><code>CREATE OR REPLACE ROLE name [AS COPY OF otherName]</code></td>
<td>Creates a new role, or if a role with the same name exists, replace it.</td>
<td>GRANT CREATE ROLE and GRANT DROP ROLE</td>
<td></td>
</tr>
<tr>
<td>RENAME ROLE</td>
<td><code>RENAME ROLE name [IF EXISTS] TO otherName</code></td>
<td>Changes the name of a role.</td>
<td>GRANT RENAME ROLE</td>
<td></td>
</tr>
<tr>
<td>DROP ROLE</td>
<td><code>DROP ROLE name [IF EXISTS]</code></td>
<td>Removes a role.</td>
<td>GRANT DROP ROLE</td>
<td></td>
</tr>
</tbody>
</table>
### Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANT ROLE(S) name[, ...] TO user[, ...]</td>
<td>Assigns roles to users. For more information, see Assigning roles to users.</td>
</tr>
<tr>
<td>REVOKE ROLE ROLE(S) name[, ...] FROM user[, ...]</td>
<td>Removes roles from users. For more information, see Revoking roles from users.</td>
</tr>
</tbody>
</table>

### Required privilege

<table>
<thead>
<tr>
<th>Command</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANT ASSIGN ROLE</td>
<td>(see DBMS ROLE MANAGEMENT privileges)</td>
</tr>
<tr>
<td>GRANT REMOVE ROLE</td>
<td>(see DBMS ROLE MANAGEMENT privileges)</td>
</tr>
</tbody>
</table>

**27.3.2. Listing roles**

Available roles can be seen using `SHOW ROLES`:

```
SHOW ROLES
```

This is the same command as `SHOW ALL ROLES`.

When first starting a Neo4j DBMS, there are a number of built-in roles:

- **PUBLIC** - a role that all users have granted. By default it gives access to the home database and to execute privileges for procedures and functions.
- **reader** - can perform traverse and read operations in all databases except `system`.
- **editor** - can perform traverse, read, and write operations in all databases except `system`, but cannot create new labels or relationship types.
- **publisher** - can do the same as `editor`, but also create new labels and relationship types.
- **architect** - can do the same as `publisher` as well as create and manage indexes and constraints.
- **admin** - can do the same as all the above, as well as manage databases, aliases, users, roles, and privileges.

<table>
<thead>
<tr>
<th>Table 496. Result</th>
</tr>
</thead>
</table>
More information about the built-in roles can be found in Operations Manual → Built-in roles.

There are multiple versions of this command, the default being SHOW ALL ROLES. To only show roles that are assigned to users, the command is SHOW POPULATED ROLES. To see which users are assigned to roles, WITH USERS can be added to the command. This will give a result with one row for each user, so if a role is assigned to two users, then it will show up twice.

SHOW POPULATED ROLES WITH USERS

The table of results will show information about the role and what database it belongs to:

Table 497. Result

<table>
<thead>
<tr>
<th>role</th>
<th>member</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;PUBLIC&quot;</td>
<td>&quot;neo4j&quot;</td>
</tr>
<tr>
<td>&quot;PUBLIC&quot;</td>
<td>&quot;bob&quot;</td>
</tr>
<tr>
<td>&quot;PUBLIC&quot;</td>
<td>&quot;user1&quot;</td>
</tr>
<tr>
<td>&quot;PUBLIC&quot;</td>
<td>&quot;user2&quot;</td>
</tr>
<tr>
<td>&quot;PUBLIC&quot;</td>
<td>&quot;user3&quot;</td>
</tr>
<tr>
<td>&quot;admin&quot;</td>
<td>&quot;neo4j&quot;</td>
</tr>
</tbody>
</table>

Rows: 6

It is also possible to filter and sort the results by using YIELD, ORDER BY and WHERE:

SHOW ROLES YIELD role ORDER BY role WHERE role ENDS WITH 'r'

In this example:

- The results have been filtered to only return the roles ending in 'r'.
- The results are ordered by the action column using ORDER BY.

It is also possible to use SKIP and LIMIT to paginate the results.

Table 498. Result
27.3.3. Creating roles

Roles can be created using `CREATE ROLE`:

```
CREATE ROLE name [IF NOT EXISTS] [AS COPY OF otherName]
```

Roles can be created or replaced by using `CREATE OR REPLACE ROLE`:

```
CREATE OR REPLACE ROLE name [AS COPY OF otherName]
```

The following naming rules apply:

- The first character must be an ASCII alphabetic character.
- Subsequent characters can be ASCII alphabetic, numeric characters, and underscore.
- Role names are case sensitive.

A role can be copied, keeping its privileges, using `CREATE ROLE name AS COPY OF otherName`.

Example 12. Copy a role

```
CREATE ROLE mysecondrole AS COPY OF myrole
```

Created roles will appear on the list provided by `SHOW ROLES`.
Example 13. List roles

SHOW ROLES

Table 499. Result

<table>
<thead>
<tr>
<th>role</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;PUBLIC&quot;</td>
</tr>
<tr>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;architect&quot;</td>
</tr>
<tr>
<td>&quot;editor&quot;</td>
</tr>
<tr>
<td>&quot;myrole&quot;</td>
</tr>
<tr>
<td>&quot;mysecondrole&quot;</td>
</tr>
<tr>
<td>&quot;publisher&quot;</td>
</tr>
<tr>
<td>&quot;reader&quot;</td>
</tr>
</tbody>
</table>

Rows: 8

The \texttt{CREATE ROLE} command is optionally idempotent, with the default behavior to throw an exception if the role already exists. Adding \texttt{IF NOT EXISTS} to the \texttt{CREATE ROLE} command will ensure that no exception is thrown and nothing happens should the role already exist.

Example 14. Create role if not exists

\begin{verbatim}
CREATE ROLE myrole IF NOT EXISTS
\end{verbatim}

The \texttt{CREATE OR REPLACE ROLE} command will result in any existing role being deleted and a new one created.

Example 15. Create or replace role

\begin{verbatim}
CREATE OR REPLACE ROLE myrole
\end{verbatim}

This is equivalent to running \texttt{DROP ROLE myrole IF EXISTS} followed by \texttt{CREATE ROLE myrole}.

- The \texttt{CREATE OR REPLACE ROLE} command does not allow you to use the \texttt{IF NOT EXISTS}.

27.3.4. Renaming roles

Roles can be renamed using \texttt{RENAME ROLE} command:
RENAMe ROLE mysecondrole TO mythirdrole

SHOW ROLES

Table 500. Result

<table>
<thead>
<tr>
<th>role</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>PUBLIC</em></td>
</tr>
<tr>
<td><em>admin</em></td>
</tr>
<tr>
<td><em>architect</em></td>
</tr>
<tr>
<td><em>editor</em></td>
</tr>
<tr>
<td><em>myrole</em></td>
</tr>
<tr>
<td><em>mythirdrole</em></td>
</tr>
<tr>
<td><em>publisher</em></td>
</tr>
<tr>
<td><em>reader</em></td>
</tr>
</tbody>
</table>

Rows: 8

The RENAME ROLE command is only available when using native authentication and authorization.

27.3.5. Assigning roles to users

Users can be given access rights by assigning them roles using GRANT ROLE:

GRANT ROLE myrole TO bob

The roles assigned to each user can be seen on the list provided by SHOW USERS:

SHOW ROLES

Table 501. Result

<table>
<thead>
<tr>
<th>user</th>
<th>roles</th>
<th>passwordChangeRequired</th>
<th>suspended</th>
<th>home</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;bob&quot;</td>
<td>[&quot;myrole&quot;,&quot;PUBLIC&quot;]</td>
<td>false</td>
<td>false</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;neo4j&quot;</td>
<td>[&quot;admin&quot;,&quot;PUBLIC&quot;]</td>
<td>true</td>
<td>false</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;user1&quot;</td>
<td>[&quot;PUBLIC&quot;]</td>
<td>true</td>
<td>false</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;user2&quot;</td>
<td>[&quot;PUBLIC&quot;]</td>
<td>true</td>
<td>false</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;user3&quot;</td>
<td>[&quot;PUBLIC&quot;]</td>
<td>true</td>
<td>false</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

Rows: 5

It is possible to assign multiple roles to multiple users in one command:
GRANT ROLES role1, role2 TO user1, user2, user3

SHOW ROLES

Table 502. Result

<table>
<thead>
<tr>
<th>user</th>
<th>roles</th>
<th>passwordChangeRequired</th>
<th>suspended</th>
<th>home</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;bob&quot;</td>
<td>[&quot;myrole&quot;,&quot;PUBLIC&quot;]</td>
<td>false</td>
<td>false</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;neo4j&quot;</td>
<td>[&quot;admin&quot;,&quot;PUBLIC&quot;]</td>
<td>true</td>
<td>false</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;user1&quot;</td>
<td>[&quot;role1&quot;,&quot;role2&quot;,&quot;PUBLIC&quot;]</td>
<td>true</td>
<td>false</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;user2&quot;</td>
<td>[&quot;role1&quot;,&quot;role2&quot;,&quot;PUBLIC&quot;]</td>
<td>true</td>
<td>false</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;user3&quot;</td>
<td>[&quot;role1&quot;,&quot;role2&quot;,&quot;PUBLIC&quot;]</td>
<td>true</td>
<td>false</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

Rows: 5

27.3.6. Revoking roles from users

Users can lose access rights by revoking their role using REVOKE ROLE:

REVOKE ROLE myrole FROM bob

The roles revoked from users can no longer be seen on the list provided by SHOW USERS:

SHOW ROLES

Table 503. Result

<table>
<thead>
<tr>
<th>user</th>
<th>roles</th>
<th>passwordChangeRequired</th>
<th>suspended</th>
<th>home</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;bob&quot;</td>
<td>[&quot;PUBLIC&quot;]</td>
<td>false</td>
<td>false</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;neo4j&quot;</td>
<td>[&quot;admin&quot;,&quot;PUBLIC&quot;]</td>
<td>true</td>
<td>false</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;user1&quot;</td>
<td>[&quot;role1&quot;,&quot;role2&quot;,&quot;PUBLIC&quot;]</td>
<td>true</td>
<td>false</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;user2&quot;</td>
<td>[&quot;role1&quot;,&quot;role2&quot;,&quot;PUBLIC&quot;]</td>
<td>true</td>
<td>false</td>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>&quot;user3&quot;</td>
<td>[&quot;role1&quot;,&quot;role2&quot;,&quot;PUBLIC&quot;]</td>
<td>true</td>
<td>false</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

Rows: 5

It is possible to revoke multiple roles from multiple users in one command:

REVOKE ROLES role1, role2 FROM user1, user2, user3

27.3.7. Deleting roles

Roles can be deleted using DROP ROLE command:
When a role has been deleted, it will no longer appear on the list provided by `SHOW ROLES`:

```
DROP ROLE mythirdrole
```

Table 504. Result

<table>
<thead>
<tr>
<th>role</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;PUBLIC&quot;</td>
</tr>
<tr>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;architect&quot;</td>
</tr>
<tr>
<td>&quot;editor&quot;</td>
</tr>
<tr>
<td>&quot;myrole&quot;</td>
</tr>
<tr>
<td>&quot;publisher&quot;</td>
</tr>
<tr>
<td>&quot;reader&quot;</td>
</tr>
</tbody>
</table>

Rows: 8

This command is optionally idempotent, with the default behavior to throw an exception if the role does not exist. Adding `IF EXISTS` to the command will ensure that no exception is thrown and nothing happens should the role not exist:

```
DROP ROLE mythirdrole IF EXISTS
```

### 27.4. Managing privileges

This section explains how to use Cypher to manage privileges for Neo4j role-based access control and fine-grained security.

Privileges control the access rights to graph elements using a combined allowlist/denylist mechanism. It is possible to grant or deny access, or use a combination of the two. The user will be able to access the resource if they have a `GRANT` (allowlist) and do not have a `DENY` (denylist) relevant to that resource. All other combinations of `GRANT` and `DENY` will result in the matching path being inaccessible. What this means in practice depends on whether we are talking about a read privilege or a write privilege:

- If an entity is not accessible due to read privileges, the data will become invisible. It will appear to the user as if they had a smaller database (smaller graph).
- If an entity is not accessible due to write privileges, an error will occur on any attempt to write that data.
In this document we will often use the terms 'allows' and 'enables' in seemingly identical ways. However, there is a subtle difference. We will use 'enables' to refer to the consequences of read privileges where a restriction will not cause an error, only a reduction in the apparent graph size. We will use 'allows' to refer to the consequence of write privileges where a restriction can result in an error.

If a user was not also provided with the database ACCESS privilege, then access to the entire database will be denied. Information about the database access privilege can be found in The ACCESS privilege.

27.4.1. Graph privilege commands (GRANT, DENY and REVOKE)

Administrators can use Cypher commands to manage Neo4j graph administrative rights. The components of the graph privilege commands are:

- the command:
  - GRANT – gives privileges to roles.
  - DENY – denies privileges to roles.
  - REVOKE – removes granted or denied privileges from roles.

- graph-privilege
  - Can be either a read privilege or write privilege.

- name
  - The graph or graphs to associate the privilege with. Because in Neo4j 4.4 you can have only one graph per database, this command uses the database name or alias to refer to that graph. When using an alias, the command will be executed on the resolved graph.

  - It can be *, which means all graphs. Graphs created after this command execution will also be associated with these privileges.

  - HOME GRAPH refers to the graph associated with the home database for that user. The default database will be used as home database if a user does not have one configured. If the user’s home database changes for any reason after privileges have been created, then these privileges will be associated with the graph attached to the new database. This can be quite powerful as it allows permissions to be switched from one graph to another simply by changing a user’s home database.

- entity
  - The graph elements this privilege applies to:
    - NODES label (nodes with the specified label(s)).
    - RELATIONSHIPS type (relationships of the specific type(s)).
ELEMENTS label (both nodes and relationships).

- The label or type can be referred with *, which means all labels or types.
- Multiple labels or types can be specified, comma-separated.
- Defaults to ELEMENTS * if omitted.
- Some of the commands for write privileges do not allow an entity part. See Write privileges for details.

- role[, ...]
  - The role or roles to associate the privilege with, comma-separated.

Table 505. General graph privilege command syntax

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANT graph-privilege ON {HOME GRAPH</td>
<td>GRAPH[S] (*</td>
</tr>
<tr>
<td>DENY graph-privilege ON {HOME GRAPH</td>
<td>GRAPH[S] (*</td>
</tr>
<tr>
<td>REVOKE GRANT graph-privilege ON {HOME GRAPH</td>
<td>GRAPH[S] (*</td>
</tr>
<tr>
<td>REVOKE DENY graph-privilege ON {HOME GRAPH</td>
<td>GRAPH[S] (*</td>
</tr>
<tr>
<td>REVOKE graph-privilege ON {HOME GRAPH</td>
<td>GRAPH[S] (*</td>
</tr>
</tbody>
</table>

DENY does NOT erase a granted privilege; they both exist. Use REVOKE if you want to remove a privilege.

The general GRANT and DENY syntaxes are illustrated in the following image:

Figure 1. GRANT and DENY Syntax
A more detailed syntax illustration for graph privileges would be the following:

![Syntax Diagram]

Figure 2. Syntax of GRANT and DENY Graph Privileges. The { and } are part of the syntax and not used for grouping.

The following image shows the hierarchy between different graph privileges:

![Hierarchy Diagram]

Figure 3. Graph privileges hierarchy

27.4.2. Listing privileges

Available privileges can be displayed using the different `SHOW PRIVILEGE[S]` commands.

Table 506. Show privileges command syntax

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`SHOW [ALL] PRIVILEGE[S] [AS [REVOKE] COMMAND[S]] [YIELD { *</td>
<td>field[, ...] } [ORDER BY field[, ...]] [SKIP n] [LIMIT n]] [WHERE expression] [RETURN field[, ...] [ORDER BY field[, ...]] [SKIP n] [LIMIT n]]`</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>SHOW ROLE[S] name[, ...] PRIVILEGE[S] [AS [REVOKE] COMMAND[S]] [YIELD { *</td>
<td>field[, ...] } [ORDER BY field[, ...]] [SKIP n] [LIMIT n]] [WHERE expression] [RETURN field[, ...] [ORDER BY field[, ...]] [SKIP n] [LIMIT n]]</td>
</tr>
</tbody>
</table>

| SHOW USER[S] [name[, ...]] PRIVILEGE[S] [AS [REVOKE] COMMAND[S]] [YIELD { * | field[, ...] } [ORDER BY field[, ...]] [SKIP n] [LIMIT n]] [WHERE expression] [RETURN field[, ...] [ORDER BY field[, ...]] [SKIP n] [LIMIT n]] | Lists privileges for a specific user, or the current user. |

Please note that it is only possible for a user to show their own privileges. Therefore, if a non-native auth provider like LDAP is in use, `SHOW USER PRIVILEGES` will only work in a limited capacity. Other users' privileges cannot be listed when using a non-native auth provider.

When using the `RETURN` clause, the `YIELD` clause is mandatory and must not be omitted.

For an easy overview of the existing privileges, it is recommended to use the `AS COMMANDS` version of the `SHOW` command. This returns the privileges as the commands that are granted or denied.

When omitting the `AS COMMANDS` clause, results will include multiple columns describing privileges:

- **access**: whether the privilege is granted or denied.
- **action**: which type of privilege this is, for example traverse, read, index management or role management.
- **resource**: what type of scope this privilege applies to, i.e. the entire DBMS, a specific database, a graph or sub-graph access.
- **graph**: the specific database or graph this privilege applies to.
- **segment**: when applicable, this privilege applies to labels, relationship types, procedures, functions or transactions.
- **role**: the role a privilege is granted to.
Examples for listing all privileges

Available privileges can be displayed using the different `SHOW PRIVILEGE[S]` commands.

**Command syntax**

```
SHOW [ALL] PRIVILEGE[S] [AS (REVOKE) COMMAND[S]]
    [WHERE expression]

SHOW [ALL] PRIVILEGE[S] [AS (REVOKE) COMMAND[S]]
    YIELD { * | field[, ...] } [ORDER BY field[, ...]] [SKIP n] [LIMIT n]
    [WHERE expression]
    [RETURN field[, ...]] [ORDER BY field[, ...]] [SKIP n] [LIMIT n]
```

**SHOW PRIVILEGES**

Lists all privileges for all roles:

**Table 507. Result**

<table>
<thead>
<tr>
<th>access</th>
<th>action</th>
<th>resource</th>
<th>graph</th>
<th>segment</th>
<th>role</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;execute&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;FUNCTION(*)&quot;</td>
<td>&quot;PUBLIC&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;execute&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;PROCEDURE(*)&quot;</td>
<td>&quot;PUBLIC&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;access&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;DEFAULT&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;PUBLIC&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;match&quot;</td>
<td>&quot;all_properties&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;NODE(*)&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;write&quot;</td>
<td>&quot;graph&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;NODE(*)&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;match&quot;</td>
<td>&quot;all_properties&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;RELATIONSHIP(*)&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;write&quot;</td>
<td>&quot;graph&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;RELATIONSHIP(*)&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;transaction_management&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;USER(*)&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;access&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;constraint&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;dbms_actions&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;index&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;start_database&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;stop_database&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;token&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;match&quot;</td>
<td>&quot;all_properties&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;NODE(*)&quot;</td>
<td>&quot;architect&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;write&quot;</td>
<td>&quot;graph&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;NODE(*)&quot;</td>
<td>&quot;architect&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;match&quot;</td>
<td>&quot;all_properties&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;RELATIONSHIP(*)&quot;</td>
<td>&quot;architect&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;write&quot;</td>
<td>&quot;graph&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;RELATIONSHIP(*)&quot;</td>
<td>&quot;architect&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;access&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;architect&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;constraint&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;architect&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;index&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;architect&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;token&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;architect&quot;</td>
</tr>
</tbody>
</table>
It is also possible to filter and sort the results by using `YIELD`, `ORDER BY` and `WHERE`:

```
SHOW PRIVILEGES YIELD role, access, action, segment ORDER BY action WHERE role = 'admin'
```

In this example:

- The number of columns returned has been reduced with the `YIELD` clause.
- The order of the returned columns has been changed.
- The results have been filtered to only return the `admin` role using a `WHERE` clause.
- The results are ordered by the `action` column using `ORDER BY`.

`SKIP` and `LIMIT` can also be used to paginate the results.

Table 508. Result

<table>
<thead>
<tr>
<th>role</th>
<th>access</th>
<th>action</th>
<th>segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;admin&quot;</td>
<td>&quot;GRANTED&quot;</td>
<td>&quot;access&quot;</td>
<td>&quot;database&quot;</td>
</tr>
<tr>
<td>&quot;admin&quot;</td>
<td>&quot;GRANTED&quot;</td>
<td>&quot;constraint&quot;</td>
<td>&quot;database&quot;</td>
</tr>
<tr>
<td>&quot;admin&quot;</td>
<td>&quot;GRANTED&quot;</td>
<td>&quot;dbms_actions&quot;</td>
<td>&quot;database&quot;</td>
</tr>
<tr>
<td>&quot;admin&quot;</td>
<td>&quot;GRANTED&quot;</td>
<td>&quot;index&quot;</td>
<td>&quot;database&quot;</td>
</tr>
<tr>
<td>&quot;admin&quot;</td>
<td>&quot;GRANTED&quot;</td>
<td>&quot;match&quot;</td>
<td>&quot;NODE(*)&quot;</td>
</tr>
</tbody>
</table>
WHERE can also be used without YIELD:

SHOW PRIVILEGES WHERE graph <> 'x'

In this example, the WHERE clause is used to filter privileges down to those that target specific graphs only.

Table 509. Result

<table>
<thead>
<tr>
<th>access</th>
<th>action</th>
<th>graph</th>
<th>resource</th>
<th>role</th>
<th>segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>access</td>
<td>&quot;DEFAULT&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;PUBLIC&quot;</td>
<td>&quot;database&quot;</td>
</tr>
<tr>
<td>&quot;DENIED&quot;</td>
<td>access</td>
<td>&quot;neo4j&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;noAccessUsers&quot;</td>
<td>&quot;database&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>access</td>
<td>&quot;neo4j&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;regularUsers&quot;</td>
<td>&quot;database&quot;</td>
</tr>
</tbody>
</table>

Rows: 3

Aggregations in the RETURN clause can be used to group privileges. In this case, by user and GRANTED or DENIED:

SHOW PRIVILEGES YIELD * RETURN role, access, collect(["graph", "resource", "segment", "action"]) as privileges

Table 510. Result

<table>
<thead>
<tr>
<th>role</th>
<th>access</th>
<th>privileges</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;PUBLIC&quot;</td>
<td>&quot;GRANTED&quot;</td>
<td>[&quot;<em>&quot; , &quot;database&quot;, &quot;FUNCTION(</em>)&quot;, &quot;execute&quot;], [&quot;<em>&quot; , &quot;database&quot;, &quot;PROCEDURE(</em>)&quot;, &quot;execute&quot;], [&quot;DEFAULT&quot;, &quot;database&quot;, &quot;database&quot;, &quot;access&quot;]]</td>
</tr>
</tbody>
</table>
The `RETURN` clause can also be used to order and paginate the results, which is useful when combined with `YIELD` and `WHERE`. In this example the query returns privileges for display five-per-page, and skips the first five to display the second page.

```cypher
SHOW PRIVILEGES YIELD * RETURN * ORDER BY role SKIP 5 LIMIT 5
```

### Table 511. Result

<table>
<thead>
<tr>
<th>access</th>
<th>action</th>
<th>graph</th>
<th>resource</th>
<th>role</th>
<th>segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;match&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;all_properties&quot;</td>
<td>&quot;admin&quot;</td>
<td>&quot;RELATIONSHIP(*)&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;write&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;graph&quot;</td>
<td>&quot;admin&quot;</td>
<td>&quot;RELATIONSHIP(*)&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;transaction_management&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;admin&quot;</td>
<td>&quot;USER(*)&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;access&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;admin&quot;</td>
<td>&quot;database&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;constraint&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;admin&quot;</td>
<td>&quot;database&quot;</td>
</tr>
</tbody>
</table>

Rows: 5

Available privileges can also be displayed as Cypher commands by adding `AS COMMAND[S]`:

```cypher
SHOW PRIVILEGES AS COMMANDS
```

### Table 512. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;DENY ACCESS ON DATABASE neo4j TO <code>noAccessUsers</code>&quot;</td>
</tr>
<tr>
<td>&quot;GRANT ACCESS ON DATABASE * TO <code>admin</code>&quot;</td>
</tr>
<tr>
<td>Command</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>&quot;GRANT ACCESS ON DATABASE * TO 'architect'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT ACCESS ON DATABASE * TO 'editor'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT ACCESS ON DATABASE * TO 'publisher'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT ACCESS ON DATABASE * TO 'reader'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT ACCESS ON DATABASE <code>neo4j</code> TO 'regularUsers'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT ACCESS ON HOME DATABASE TO 'PUBLIC'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT ALL DBMS PRIVILEGES ON DBMS TO 'admin'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT CONSTRAINT MANAGEMENT ON DATABASE * TO 'admin'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT CONSTRAINT MANAGEMENT ON DATABASE * TO 'architect'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT EXECUTE FUNCTION * ON DBMS TO 'PUBLIC'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT EXECUTE PROCEDURE * ON DBMS TO 'PUBLIC'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT INDEX MANAGEMENT ON DATABASE * TO 'admin'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT INDEX MANAGEMENT ON DATABASE * TO 'architect'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT MATCH (*) ON GRAPH * NODE * TO 'admin'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT MATCH (*) ON GRAPH * NODE * TO 'architect'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT MATCH (*) ON GRAPH * NODE * TO 'editor'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT MATCH (*) ON GRAPH * NODE * TO 'publisher'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT MATCH (*) ON GRAPH * NODE * TO 'reader'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT MATCH (*) ON GRAPH * RELATIONSHIP * TO 'admin'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT MATCH (*) ON GRAPH * RELATIONSHIP * TO 'architect'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT MATCH (*) ON GRAPH * RELATIONSHIP * TO 'editor'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT MATCH (*) ON GRAPH * RELATIONSHIP * TO 'publisher'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT MATCH (*) ON GRAPH * RELATIONSHIP * TO 'reader'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT NAME MANAGEMENT ON DATABASE * TO 'admin'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT NAME MANAGEMENT ON DATABASE * TO 'architect'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT NAME MANAGEMENT ON DATABASE * TO 'publisher'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT START ON DATABASE * TO 'admin'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT STOP ON DATABASE * TO 'admin'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT TRANSACTION MANAGEMENT (*) ON DATABASE * TO 'admin'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT WRITE ON GRAPH * TO 'admin'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT WRITE ON GRAPH * TO 'architect'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT WRITE ON GRAPH * TO 'editor'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT WRITE ON GRAPH * TO 'publisher'&quot;</td>
</tr>
</tbody>
</table>

Rows: 35

Like other SHOW commands, the output can also be processed using **YIELD / WHERE / RETURN**.
SHOW PRIVILEGES AS COMMANDS WHERE command CONTAINS 'MANAGEMENT'

Table 513. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT CONSTRAINT MANAGEMENT ON DATABASE * TO 'admin'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT CONSTRAINT MANAGEMENT ON DATABASE * TO 'architect'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT INDEX MANAGEMENT ON DATABASE * TO 'admin'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT INDEX MANAGEMENT ON DATABASE * TO 'architect'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT NAME MANAGEMENT ON DATABASE * TO 'admin'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT NAME MANAGEMENT ON DATABASE * TO 'architect'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT NAME MANAGEMENT ON DATABASE * TO 'publisher'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT TRANSACTION MANAGEMENT (*) ON DATABASE * TO 'admin'&quot;</td>
</tr>
</tbody>
</table>

Rows: 8

It is also possible to get the privileges listed as revoking commands instead of granting or denying:

SHOW PRIVILEGES AS REVOKE COMMANDS

Table 514. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;REVOKE DENY ACCESS ON DATABASE neo4j FROM 'noAccessUsers'&quot;</td>
</tr>
<tr>
<td>&quot;REVOKE GRANT ACCESS ON DATABASE * FROM 'admin'&quot;</td>
</tr>
<tr>
<td>&quot;REVOKE GRANT ACCESS ON DATABASE * FROM 'architect'&quot;</td>
</tr>
<tr>
<td>&quot;REVOKE GRANT ACCESS ON DATABASE * FROM 'editor'&quot;</td>
</tr>
<tr>
<td>&quot;REVOKE GRANT ACCESS ON DATABASE * FROM 'publisher'&quot;</td>
</tr>
<tr>
<td>&quot;REVOKE GRANT ACCESS ON DATABASE * FROM 'reader'&quot;</td>
</tr>
<tr>
<td>&quot;REVOKE GRANT ACCESS ON DATABASE neo4j FROM 'regularUsers'&quot;</td>
</tr>
<tr>
<td>&quot;REVOKE GRANT ACCESS ON HOME DATABASE FROM 'PUBLIC'&quot;</td>
</tr>
<tr>
<td>&quot;REVOKE GRANT ALL DBMS PRIVILEGES ON DBMS FROM 'admin'&quot;</td>
</tr>
<tr>
<td>&quot;REVOKE GRANT CONSTRAINT MANAGEMENT ON DATABASE * FROM 'admin'&quot;</td>
</tr>
<tr>
<td>&quot;REVOKE GRANT CONSTRAINT MANAGEMENT ON DATABASE * FROM 'architect'&quot;</td>
</tr>
<tr>
<td>&quot;REVOKE GRANT EXECUTE FUNCTION * ON DBMS FROM 'PUBLIC'&quot;</td>
</tr>
<tr>
<td>&quot;REVOKE GRANT EXECUTE PROCEDURE * ON DBMS FROM 'PUBLIC'&quot;</td>
</tr>
<tr>
<td>&quot;REVOKE GRANT INDEX MANAGEMENT ON DATABASE * FROM 'admin'&quot;</td>
</tr>
<tr>
<td>&quot;REVOKE GRANT INDEX MANAGEMENT ON DATABASE * FROM 'architect'&quot;</td>
</tr>
<tr>
<td>&quot;REVOKE GRANT MATCH {*} ON GRAPH * NODE * FROM 'admin'&quot;</td>
</tr>
<tr>
<td>&quot;REVOKE GRANT MATCH {*} ON GRAPH * NODE * FROM 'architect'&quot;</td>
</tr>
<tr>
<td>&quot;REVOKE GRANT MATCH {*} ON GRAPH * NODE * FROM 'editor'&quot;</td>
</tr>
</tbody>
</table>
command

"REVOKE GRANT MATCH (*) ON GRAPH * NODE * FROM 'publisher''
"REVOKE GRANT MATCH (*) ON GRAPH * NODE * FROM 'reader''
"REVOKE GRANT MATCH (*) ON GRAPH * RELATIONSHIP * FROM 'admin''
"REVOKE GRANT MATCH (*) ON GRAPH * RELATIONSHIP * FROM 'architect''
"REVOKE GRANT MATCH (*) ON GRAPH * RELATIONSHIP * FROM 'editor''
"REVOKE GRANT MATCH (*) ON GRAPH * RELATIONSHIP * FROM 'publisher''
"REVOKE GRANT MATCH (*) ON GRAPH * RELATIONSHIP * FROM 'reader''
"REVOKE GRANT NAME MANAGEMENT ON DATABASE * FROM 'admin''
"REVOKE GRANT NAME MANAGEMENT ON DATABASE * FROM 'architect''
"REVOKE GRANT NAME MANAGEMENT ON DATABASE * FROM 'publisher''
"REVOKE GRANT START ON DATABASE * FROM 'admin''
"REVOKE GRANT STOP ON DATABASE * FROM 'admin''
"REVOKE GRANT TRANSACTION MANAGEMENT (*) ON DATABASE * FROM 'admin''
"REVOKE GRANT WRITE ON GRAPH * FROM 'admin''
"REVOKE GRANT WRITE ON GRAPH * FROM 'architect''
"REVOKE GRANT WRITE ON GRAPH * FROM 'editor''
"REVOKE GRANT WRITE ON GRAPH * FROM 'publisher''

Rows: 35

For more info about revoking privileges, please see the REVOKE command.

Examples for listing privileges for specific roles **Enterprise edition**

Available privileges for specific roles can be displayed using **SHOW ROLE** name **PRIVILEGE[S]**:

SHOW ROLE[S] name[, ...] PRIVILEGE[S] [AS [REVOKE] COMMAND[S]]
   [WHERE expression]

SHOW ROLE[S] name[, ...] PRIVILEGE[S] [AS [REVOKE] COMMAND[S]]
   [WHERE expression]
   [YIELD ( * | field[, ...] ) [ORDER BY field[, ...]] [SKIP n] [LIMIT n]
   [WHERE expression]
   [RETURN field[, ...] [ORDER BY field[, ...]] [SKIP n] [LIMIT n]]

SHOW ROLE regularUsers PRIVILEGES

Lists all privileges for role **regularUsers**.

Table 515. Result

<table>
<thead>
<tr>
<th>access</th>
<th>action</th>
<th>graph</th>
<th>resource</th>
<th>role</th>
<th>segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;access&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;neo4j&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;regularUsers&quot;</td>
</tr>
</tbody>
</table>

483
SHOW ROLES regularUsers, noAccessUsers PRIVILEGES

Lists all privileges for roles regularUsers and noAccessUsers.

Table 516. Result

<table>
<thead>
<tr>
<th>access</th>
<th>action</th>
<th>graph</th>
<th>resource</th>
<th>role</th>
<th>segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;DENIED&quot;</td>
<td>&quot;access&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;neo4j&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;noAccessUsers&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;access&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;neo4j&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;regularUsers&quot;</td>
</tr>
</tbody>
</table>

Rows: 2

Similar to the other SHOW PRIVILEGES commands, the available privileges for roles can also be listed as Cypher commands with the optional AS COMMAND[S].

Table 517. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT ACCESS ON DATABASE * TO <code>admin</code>&quot;</td>
</tr>
<tr>
<td>&quot;GRANT ALL DBMS PRIVILEGES ON DBMS TO <code>admin</code>&quot;</td>
</tr>
<tr>
<td>&quot;GRANT CONSTRAINT MANAGEMENT ON DATABASE * TO <code>admin</code>&quot;</td>
</tr>
<tr>
<td>&quot;GRANT INDEX MANAGEMENT ON DATABASE * TO <code>admin</code>&quot;</td>
</tr>
<tr>
<td>&quot;GRANT MATCH (*) ON GRAPH * NODE * TO <code>admin</code>&quot;</td>
</tr>
<tr>
<td>&quot;GRANT MATCH (*) ON GRAPH * RELATIONSHIP * TO <code>admin</code>&quot;</td>
</tr>
<tr>
<td>&quot;GRANT NAME MANAGEMENT ON DATABASE * TO <code>admin</code>&quot;</td>
</tr>
<tr>
<td>&quot;GRANT START ON DATABASE * TO <code>admin</code>&quot;</td>
</tr>
<tr>
<td>&quot;GRANT STOP ON DATABASE * TO <code>admin</code>&quot;</td>
</tr>
<tr>
<td>&quot;GRANT TRANSACTION MANAGEMENT (*) ON DATABASE * TO <code>admin</code>&quot;</td>
</tr>
<tr>
<td>&quot;GRANT WRITE ON GRAPH * TO <code>admin</code>&quot;</td>
</tr>
</tbody>
</table>

Rows: 11

The output can be processed using YIELD / WHERE / RETURN here as well:

SHOW ROLE architect PRIVILEGES AS COMMANDS WHERE command CONTAINS 'MATCH'

Table 518. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT MATCH (*) ON GRAPH * NODE * TO <code>architect</code>&quot;</td>
</tr>
</tbody>
</table>
Again, it is possible to get the privileges listed as revoking commands instead of granting or denying. For more info about revoking privileges, please see the REVOKE command.

SHOW ROLE reader PRIVILEGES AS REVOKE COMMANDS

Table 519. Result

command

"REVOKE GRANT ACCESS ON DATABASE * FROM 'reader';"  
"REVOKE GRANT MATCH {*} ON GRAPH * NODE * FROM 'reader';"  
"REVOKE GRANT MATCH {*} ON GRAPH * RELATIONSHIP * FROM 'reader';"

Rows: 3

Examples for listing privileges for specific users [Enterprise edition]

Available privileges for specific users can be displayed using SHOW USER name PRIVILEGES.

SHOW USER[s] [name[, ...]] PRIVILEGE[s] [AS REVOKE COMMAND[s]]  
[WHERE expression]

Table 520. Result

<table>
<thead>
<tr>
<th>access</th>
<th>action</th>
<th>resource</th>
<th>graph</th>
<th>resource</th>
<th>role</th>
<th>segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;execute&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;FUNCTION(*)&quot;</td>
<td>&quot;PUBLIC&quot;</td>
<td>&quot;jake&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;execute&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;PROCEDURE(*)&quot;</td>
<td>&quot;PUBLIC&quot;</td>
<td>&quot;jake&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;access&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;DEFAULT&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;PUBLIC&quot;</td>
<td>&quot;jake&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;access&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;neo4j&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;regularUsers&quot;</td>
<td>&quot;jake&quot;</td>
</tr>
</tbody>
</table>
SHOW USERS jake, joe PRIVILEGES

Lists all privileges for users jake and joe.

Table 521. Result

<table>
<thead>
<tr>
<th>access</th>
<th>action</th>
<th>resource</th>
<th>graph</th>
<th>resource</th>
<th>role</th>
<th>segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;execute&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;FUNCTION(*)&quot;</td>
<td>&quot;PUBLIC&quot;</td>
<td>&quot;jake&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;execute&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;PROCEDURE(*)&quot;</td>
<td>&quot;PUBLIC&quot;</td>
<td>&quot;jake&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;access&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;DEFAULT&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;PUBLIC&quot;</td>
<td>&quot;jake&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;access&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;neo4j&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;regularUsers&quot;</td>
<td>&quot;jake&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;execute&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;FUNCTION(*)&quot;</td>
<td>&quot;PUBLIC&quot;</td>
<td>&quot;joe&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;execute&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;PROCEDURE(*)&quot;</td>
<td>&quot;PUBLIC&quot;</td>
<td>&quot;joe&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;access&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;DEFAULT&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;PUBLIC&quot;</td>
<td>&quot;joe&quot;</td>
</tr>
<tr>
<td>&quot;DENIED&quot;</td>
<td>&quot;access&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;neo4j&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;noAccessUsers&quot;</td>
<td>&quot;joe&quot;</td>
</tr>
</tbody>
</table>

The same command can be used at all times to review available privileges for the current user. For this purpose, there is a shorter form of the command: SHOW USER PRIVILEGES:

SHOW USER PRIVILEGES

As for the other privilege commands, available privileges for users can also be listed as Cypher commands with the optional AS COMMAND[S].

When showing user privileges as commands, the roles in the Cypher commands are replaced with a parameter. This can be used to quickly create new roles based on the privileges of specific users.

SHOW USER jake PRIVILEGES AS COMMANDS

Table 522. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT ACCESS ON DATABASE neo4j TO $role&quot;</td>
</tr>
<tr>
<td>&quot;GRANT ACCESS ON HOME DATABASE TO $role&quot;</td>
</tr>
<tr>
<td>&quot;GRANT EXECUTE FUNCTION * ON DBMS TO $role&quot;</td>
</tr>
<tr>
<td>&quot;GRANT EXECUTE PROCEDURE * ON DBMS TO $role&quot;</td>
</tr>
</tbody>
</table>
Like other `SHOW` commands, the output can also be processed using `YIELD / WHERE / RETURN`. Additionally, similar to the other show privilege commands, it is also possible to show the commands for revoking the privileges.

```
SHOW USER jake PRIVILEGES AS REVOKE COMMANDS WHERE command CONTAINS 'EXECUTE'
```

Table 523. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;REVOKE GRANT EXECUTE FUNCTION * ON DBMS FROM $role&quot;</td>
</tr>
<tr>
<td>&quot;REVOKE GRANT EXECUTE PROCEDURE * ON DBMS FROM $role&quot;</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Rows: 2</td>
</tr>
</tbody>
</table>

27.4.3. Revoking privileges **Enterprise edition**

Privileges that were granted or denied earlier can be revoked using the `REVOKE` command:

```
REVOKE [ GRANT | DENY ] graph-privilege
       FROM role[, ...]
```

An example usage of the `REVOKE` command is given here:

```
REVOKE GRANT TRAVERSE ON HOME GRAPH NODES Post FROM regularUsers
```

While it can be explicitly specified that `REVOKE` should remove a `GRANT` or `DENY`, it is also possible to `REVOKE` both by not specifying them at all, as the next example demonstrates. Because of this, if there happens to be a `GRANT` and a `DENY` for the same privilege, it would remove both.

```
REVOKE TRAVERSE ON HOME GRAPH NODES Payments FROM regularUsers
```

27.5. Built-in roles and privileges **Enterprise edition**

This section explains the default privileges of the built-in roles in Neo4j and how to recreate them if needed.

All of the commands described in this chapter require that the user executing the commands has the rights to do so. The privileges listed in the following sections are the default set of privileges for each built-in role:

- The **PUBLIC** role
• The reader role
• The editor role
• The publisher role
• The architect role
• The admin role

27.5.1. The PUBLIC role

All users are granted the PUBLIC role, and it can not be revoked or dropped. By default, it gives access to the default database and allows executing all procedures and user-defined functions.

The PUBLIC role cannot be dropped or revoked from any user, but the specific privileges for the role can be modified. In contrast to the PUBLIC role, the other built-in roles can be granted, revoked, dropped, and re-created.

Listing PUBLIC role privileges

SHOW ROLE PUBLIC PRIVILEGES AS COMMANDS

Table 524. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT ACCESS ON HOME DATABASE TO 'PUBLIC'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT EXECUTE FUNCTION * ON DBMS TO 'PUBLIC'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT EXECUTE PROCEDURE * ON DBMS TO 'PUBLIC'&quot;</td>
</tr>
<tr>
<td>Rows: 3</td>
</tr>
</tbody>
</table>

Recreating the PUBLIC role

The PUBLIC role can not be dropped and thus there is no need to recreate the role itself. To restore the role to its original capabilities, two steps are needed.

First, all GRANT or DENY privileges on this role should be revoked (see output of SHOW ROLE PUBLIC PRIVILEGES AS REVOKE COMMANDS on what to revoke). Secondly, run these queries:

GRANT ACCESS ON HOME DATABASE TO PUBLIC

GRANT EXECUTE PROCEDURES * ON DBMS TO PUBLIC

GRANT EXECUTE USER DEFINED FUNCTIONS * ON DBMS TO PUBLIC

The resulting PUBLIC role now has the same privileges as the original built-in PUBLIC role.
27.5.2. The reader role

The reader role can perform read-only queries on all graphs except for the system database.

Listing reader role privileges

```
SHOW ROLE reader PRIVILEGES AS COMMANDS
```

Table 525. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT ACCESS ON DATABASE * TO <code>reader</code>&quot;</td>
</tr>
<tr>
<td>&quot;GRANT MATCH {*} ON GRAPH * NODE * TO <code>reader</code>&quot;</td>
</tr>
<tr>
<td>&quot;GRANT MATCH {*} ON GRAPH * RELATIONSHIP * TO <code>reader</code>&quot;</td>
</tr>
</tbody>
</table>

Rows: 3

Recreating the reader role

To restore the role to its original capabilities two steps are needed. First, execute DROP ROLE reader. Secondly, run these queries:

```
CREATE ROLE reader
GRANT ACCESS ON DATABASE * TO reader
GRANT MATCH {*} ON GRAPH * NODE * TO reader
```

The resulting reader role now has the same privileges as the original built-in reader role.

27.5.3. The editor role

The editor role can perform read and write operations on all graphs except for the system database, but it cannot create new labels, property keys or relationship types.

Listing editor role privileges

```
SHOW ROLE editor PRIVILEGES AS COMMANDS
```

Table 526. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT ACCESS ON DATABASE * TO <code>editor</code>&quot;</td>
</tr>
<tr>
<td>&quot;GRANT MATCH {*} ON GRAPH * NODE * TO <code>editor</code>&quot;</td>
</tr>
</tbody>
</table>

489
Recreating the `editor` role

To restore the role to its original capabilities two steps are needed. First, execute `DROP ROLE editor`. Secondly, run these queries:

```
CREATE ROLE editor
GRANT ACCESS ON DATABASE * TO editor
GRANT MATCH (*) ON GRAPH * TO editor
GRANT NAME MANAGEMENT ON DATABASE * TO `editor`
GRANT WRITE ON GRAPH * TO editor
```

The resulting `editor` role now has the same privileges as the original built-in `editor` role.

### 27.5.4. The `publisher` role

The `publisher` role can do the same as `editor`, as well as create new labels, property keys and relationship types.

**Listing `publisher` role privileges**

```
SHOW ROLE publisher PRIVILEGES AS COMMANDS
```

**Table 527. Result**

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT ACCESS ON DATABASE * TO 'publisher'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT MATCH (*) ON NODE * TO 'publisher'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT MATCH (*) ON RELATIONSHIP * TO 'publisher'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT NAME MANAGEMENT ON DATABASE * TO 'publisher'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT WRITE ON GRAPH * TO 'publisher'&quot;</td>
</tr>
</tbody>
</table>

Rows: 5
Recreating the **publisher** role

To restore the role to its original capabilities two steps are needed. First, execute `DROP ROLE publisher`. Secondly, run these queries:

```
CREATE ROLE publisher
GRANT ACCESS ON DATABASE * TO publisher
GRANT MATCH ({*}) ON GRAPH * TO publisher
GRANT WRITE ON GRAPH * TO publisher
GRANT NAME MANAGEMENT ON DATABASE * TO publisher
```

The resulting **publisher** role now has the same privileges as the original built-in **publisher** role.

27.5.5. The **architect** role

The **architect** role can do the same as the **publisher**, as well as create and manage indexes and constraints.

Listing **architect** role privileges

```
SHOW ROLE architect PRIVILEGES AS COMMANDS
```

Table 528. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT ACCESS ON DATABASE * TO &quot;architect&quot;&quot;</td>
</tr>
<tr>
<td>&quot;GRANT CONSTRAINT MANAGEMENT ON DATABASE * TO &quot;architect&quot;&quot;</td>
</tr>
<tr>
<td>&quot;GRANT INDEX MANAGEMENT ON DATABASE * TO &quot;architect&quot;&quot;</td>
</tr>
<tr>
<td>&quot;GRANT MATCH ({*}) ON GRAPH * NODE * TO &quot;architect&quot;&quot;</td>
</tr>
<tr>
<td>&quot;GRANT MATCH ({*}) ON GRAPH * RELATIONSHIP * TO &quot;architect&quot;&quot;</td>
</tr>
<tr>
<td>&quot;GRANT NAME MANAGEMENT ON DATABASE * TO &quot;architect&quot;&quot;</td>
</tr>
<tr>
<td>&quot;GRANT WRITE ON GRAPH * TO &quot;architect&quot;&quot;</td>
</tr>
</tbody>
</table>

Rows: 7

Recreating the **architect** role

To restore the role to its original capabilities two steps are needed. First, execute `DROP ROLE architect`. Secondly, run these queries: 
The resulting **architect** role now has the same privileges as the original built-in **architect** role.

### 27.5.6. The **admin** role

The **admin** role can do the same as the **architect**, as well as manage databases, aliases, users, roles and privileges.

The **admin** role has the ability to perform administrative tasks. These include the rights to perform the following classes of tasks:

- **Manage database security** to control the rights to perform actions on specific databases:
  - Manage access to a database and the right to start and stop a database.
  - Manage indexes and constraints.
  - Allow the creation of labels, relationship types or property names.
  - Manage transactions
- **Manage DBMS security** to control the rights to perform actions on the entire system:
  - Manage multiple databases.
  - Manage users and roles.
  - Change configuration parameters.
  - Manage sub-graph privileges.
  - Manage procedure security.

These rights are conferred using privileges that can be managed through the `GRANT`, `DENY` and `REVOKE` commands.

### Listing **admin** role privileges

```sql
SHOW ROLE admin PRIVILEGES AS COMMANDS
```
Table 529. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT ACCESS ON DATABASE * TO 'admin'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT ALL DBMS PRIVILEGES ON DBMS TO 'admin'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT CONSTRAINT MANAGEMENT ON DATABASE * TO 'admin'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT INDEX MANAGEMENT ON DATABASE * TO 'admin'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT MATCH {*} ON GRAPH * NODE * TO 'admin'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT MATCH {*} ON GRAPH * RELATIONSHIP * TO 'admin'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT NAME MANAGEMENT ON DATABASE * TO 'admin'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT START ON DATABASE * TO 'admin'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT STOP ON DATABASE * TO 'admin'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT TRANSACTION MANAGEMENT (*) ON DATABASE * TO 'admin'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT WRITE ON GRAPH * TO 'admin'&quot;</td>
</tr>
</tbody>
</table>

Rows: 11

If the built-in admin role has been altered or dropped, and needs to be restored to its original state, see Operations Manual → Password and user recovery.

Recreating the admin role

To restore the role to its original capabilities two steps are needed. First, execute DROP ROLE admin. Secondly, run these queries:

```sql
CREATE ROLE admin

GRANT ALL DBMS PRIVILEGES ON DBMS TO admin

GRANT TRANSACTION MANAGEMENT ON DATABASE * TO admin

GRANT START ON DATABASE * TO admin

GRANT STOP ON DATABASE * TO admin

GRANT MATCH {*} ON GRAPH * TO admin

GRANT WRITE ON GRAPH * TO admin

GRANT ALL ON DATABASE * TO admin
```
The resulting admin role now has the same privileges as the original built-in admin role.

Additional information about restoring the admin role can be found at Operations Manual → Recover the admin role.

27.6. Read privileges

This section explains how to use Cypher to manage read privileges on graphs.

There are three separate read privileges:

- **TRAVERSE** - enables the specified entities to be found.
- **READ** - enables the specified properties of the found entities to be read.
- **MATCH** - combines both **TRAVERSE** and **READ**, enabling an entity to be found and its properties read.

27.6.1. The **TRAVERSE** privilege

Users can be granted the right to find nodes and relationships using the **GRANT TRAVERSE** privilege.

```cypher
GRANT TRAVERSE
ON {HOME GRAPH | GRAPH[S] { * | name[, ...] }}
[
    ELEMENT[S] { * | label-or-rel-type[, ...] }
    | NODE[S] { * | label[, ...] }
    | RELATIONSHIP[S] { * | rel-type[, ...] }
] TO role[, ...]
```

For example, we can enable the user jake, who has the role 'regularUsers' to find all nodes with the label Post:

```cypher
GRANT TRAVERSE ON GRAPH neo4j NODES Post TO regularUsers
```

The **TRAVERSE** privilege can also be denied.

```cypher
DENY TRAVERSE
ON {HOME GRAPH | GRAPH[S] { * | name[, ...] }}
[
    ELEMENT[S] { * | label-or-rel-type[, ...] }
    | NODE[S] { * | label[, ...] }
    | RELATIONSHIP[S] { * | rel-type[, ...] }
] TO role[, ...]
```

For example, we can disable the user jake, who has the role 'regularUsers' from finding all nodes with the label Payments:

```cypher
DENY TRAVERSE ON HOME GRAPH NODES Payments TO regularUsers
```
### 27.6.2. The **READ** privilege

Users can be granted the right to do property reads on nodes and relationships using the **GRANT READ** privilege. It is very important to note that users can only read properties on entities that they are enabled to find in the first place.

```plaintext
GRANT READ
"{* { * | property[, ...] } "}
ON (HOME GRAPH | GRAPH[S] { * | name[, ...] })
[
    ELEMENT[S] { * | label-or-rel-type[, ...] } |
    NODE[S] { * | label[, ...] } |
    RELATIONSHIP[S] { * | rel-type[, ...] }
]
TO role[, ...]
```

For example, we can enable the user *jake*, who has the role 'regularUsers' to read all properties on nodes with the label *Post*. The * implies that the ability to read all properties also extends to properties that might be added in the future.

```plaintext
GRANT READ ( * ) ON GRAPH neo4j NODES Post TO regularUsers
```

Granting property **READ** access does not imply that the entities with that property can be found. For example, if there is also a **DENY TRAVERSE** present on the same entity as a **GRANT READ**, the entity will not be found by a Cypher **MATCH** statement.

The **READ** privilege can also be denied.

```plaintext
DENY READ
"{* { * | property[, ...] } "}
ON (HOME GRAPH | GRAPH[S] { * | name[, ...] })
[
    ELEMENT[S] { * | label-or-rel-type[, ...] } |
    NODE[S] { * | label[, ...] } |
    RELATIONSHIP[S] { * | rel-type[, ...] }
]
TO role[, ...]
```

Although we just granted the user *jake* the right to read all properties, we may want to hide the *secret* property. The following example shows how to do that:

```plaintext
DENY READ { secret } ON GRAPH neo4j NODES Post TO regularUsers
```

### 27.6.3. The **MATCH** privilege

Users can be granted the right to find and do property reads on nodes and relationships using the **GRANT MATCH** privilege. This is semantically the same as having both **TRAVERSE** and **READ** privileges.
27.7. Write privileges

This section explains how to use Cypher to manage write privileges on graphs.

Write privileges are defined for different parts of the graph:

- **Elements**
  - Nodes
  - Relationships

- **Properties**
  - Label
  - Type

- **Searching**
  - Name

#### GRANT MATCH

```cypher
GRANT MATCH
"{ " * | property[, ...] } "}"
ON (HOME GRAPH | GRAPH[S] { " | name[, ...]} )
[ ELEM[S] { " | label-or-rel-type[, ...]} ]
| NODE[S] { " | label[, ...]} ]
| RELATIONSHIP[S] { " | rel-type[, ...]} ]
] TO role[, ...]
```

For example, if you want to grant the ability to read the properties `language` and `length` for nodes with the label `Message`, as well as the ability to find these nodes to the role `regularUsers`, you can use the following `GRANT MATCH` query:

```cypher
GRANT MATCH { language, length } ON GRAPH neo4j NODES Message TO regularUsers
```

Like all other privileges, the `MATCH` privilege can also be denied.

#### DENY MATCH

```cypher
DENY MATCH
"{ " * | property[, ...] } "}"
ON (HOME GRAPH | GRAPH[S] { " | name[, ...]} )
[ ELEM[S] { " | label-or-rel-type[, ...]} ]
| NODE[S] { " | label[, ...]} ]
| RELATIONSHIP[S] { " | rel-type[, ...]} ]
] TO role[, ...]
```

Please note that the effect of denying a `MATCH` privilege depends on whether concrete property keys are specified or are `*`. If you specify concrete property keys, then `DENY MATCH` will only deny reading those properties. Finding the elements to traverse would still be enabled. If you specify `*` instead, then both traversal of the element and all property reads will be disabled. The following queries will show examples for this.

Denying to read the property `content` on nodes with the label `Message` for the role `regularUsers` would look like the following query. Although not being able to read this specific property, nodes with that label can still be traversed (and, depending on other grants, other properties on it could still be read).

```cypher
DENY MATCH { content } ON GRAPH neo4j NODES Message TO regularUsers
```

The following query exemplifies how it would look if you wanted to deny both reading all properties and traversing nodes labeled with `Account`:

```cypher
DENY MATCH { * } ON GRAPH neo4j NODES Account TO regularUsers
```
• **CREATE** - allows creating nodes and relationships.
• **DELETE** - allows deleting nodes and relationships.
• **SET LABEL** - allows setting the specified node labels using the **SET** clause.
• **REMOVE LABEL** - allows removing the specified node labels using the **REMOVE** clause.
• **SET PROPERTY** - allows setting properties on nodes and relationships.

There are also compound privileges which combine the above specific privileges:

• **MERGE** - allows **MATCH**, **CREATE** and **SET PROPERTY** to apply the **MERGE** command.
• **WRITE** - allows all **WRITE** operations on an entire graph.
• **ALL GRAPH PRIVILEGES** - allows all **READ** and **WRITE** operations on an entire graph.

### 27.7.1. The **CREATE** privilege

The **CREATE** privilege allows a user to create new node and relationship elements on a graph. See the Cypher **CREATE** clause.

```
GRANT CREATE ON {HOME GRAPH | GRAPH[S] { * | name[, ...] }}
 |
 ELEMENT[S] { * | label-or-rel-type[, ...] }
 |
 NODE[S] { * | label[, ...] }
 |
 RELATIONSHIP[S] { * | rel-type[, ...] }
 |
 TO role[, ...]
```

For example, to grant the role *regularUsers* the ability to **CREATE** elements on the graph *neo4j*, use:

```
GRANT CREATE ON GRAPH neo4j ELEMENTS * TO regularUsers
```

The **CREATE** privilege can also be denied:

```
DENY CREATE ON {HOME GRAPH | GRAPH[S] { * | name[, ...] }}
 |
 ELEMENT[S] { * | label-or-rel-type[, ...] }
 |
 NODE[S] { * | label[, ...] }
 |
 RELATIONSHIP[S] { * | rel-type[, ...] }
 |
 TO role[, ...]
```

For example, to deny the role *regularUsers* the ability to **CREATE** nodes with the label *foo* on all graphs, use:

```
DENY CREATE ON GRAPH * NODES foo TO regularUsers
```

If the user attempts to create nodes with a label that does not already exist on the database, then the user must also possess the **CREATE NEW LABEL** privilege. The same applies to new relationships: the **CREATE NEW RELATIONSHIP TYPE** privilege is required.
27.7.2. The **DELETE** privilege

The **DELETE** privilege allows a user to delete node and relationship elements on a graph. See the Cypher **DELETE** clause.

```cypher
GRANT DELETE ON {HOME GRAPH | GRAPH[S] { * | name[, ...] }}
    [ ELEMENT[S] { * | label-or-rel-type[, ...] }
    | NODE[S] { * | label[, ...] }
    | RELATIONSHIP[S] { * | rel-type[, ...] }
] TO role[, ...]
```

For example, to grant the role `regularUsers` the ability to **DELETE** elements on the graph `neo4j`, use:

```cypher
GRANT DELETE ON GRAPH neo4j ELEMENTS * TO regularUsers
```

The **DELETE** privilege can also be denied:

```cypher
DENY DELETE ON {HOME GRAPH | GRAPH[S] { * | name[, ...] }}
    [ ELEMENT[S] { * | label-or-rel-type[, ...] }
    | NODE[S] { * | label[, ...] }
    | RELATIONSHIP[S] { * | rel-type[, ...] }
] TO role[, ...]
```

For example, to deny the role `regularUsers` the ability to **DELETE** relationships with the relationship type `bar` on all graphs, use:

```cypher
DENY DELETE ON GRAPH * RELATIONSHIPS bar TO regularUsers
```

Users with **DELETE** privilege, but restricted **TRAVERSE** privileges, will not be able to do **DETACH DELETE** in all cases. See Operations Manual → Fine-grained access control for more info.

27.7.3. The **SET LABEL** privilege

The **SET LABEL** privilege allows you to set labels on a node by using the **SET** clause:

```cypher
GRANT SET LABEL { * | label[, ...] }
    ON {HOME GRAPH | GRAPH[S] { * | name[, ...] }}
    TO role[, ...]
```

For example, to grant the role `regularUsers` the ability to **SET** any label on nodes of the graph `neo4j`, use:

```cypher
GRANT SET LABEL * ON GRAPH neo4j TO regularUsers
```

Unlike many of the other **READ** and **WRITE** privileges, it is not possible to restrict the **SET LABEL** privilege to specific ELEMENTS, NODES or RELATIONSHIPS.
The `SET LABEL` privilege can also be denied:

```sql
DENY SET LABEL { * | label[, ...] }
   ON (HOME GRAPH | GRAPH[S] { * | name[, ...] })
   TO role[, ...]
```

For example, to deny the role `regularUsers` the ability to `SET` the label `foo` on nodes of all graphs, use:

```sql
DENY SET LABEL foo ON GRAPH * TO regularUsers
```

If no instances of this label exist on the database, then the `CREATE NEW LABEL` privilege is also required.

### 27.7.4. The `REMOVE LABEL` privilege

The `REMOVE LABEL` privilege allows you to remove labels from a node by using the `REMOVE` clause:

```sql
GRANT REMOVE LABEL { * | label[, ...] }
   ON (HOME GRAPH | GRAPH[S] { * | name[, ...] })
   TO role[, ...]
```

For example, to grant the role `regularUsers` the ability to `REMOVE` any label from nodes of the graph `neo4j`, use:

```sql
GRANT REMOVE LABEL * ON GRAPH neo4j TO regularUsers
```

Unlike many of the other `READ` and `WRITE` privileges, it is not possible to restrict the `REMOVE LABEL` privilege to specific ELEMENTS, NODES or RELATIONSHIPS.

The `REMOVE LABEL` privilege can also be denied:

```sql
DENY REMOVE LABEL { * | label[, ...] }
   ON (HOME GRAPH | GRAPH[S] { * | name[, ...] })
   TO role[, ...]
```

For example, denying the role `regularUsers` the ability to remove the label `foo` from nodes of all graphs, use:

```sql
DENY REMOVE LABEL foo ON GRAPH * TO regularUsers
```

### 27.7.5. The `SET PROPERTY` privilege

The `SET PROPERTY` privilege allows a user to set a property on a node or relationship element in a graph by using the `SET` clause:
GRANT SET PROPERTY "{" ( * | property[, ...] ) "}" ON (HOME GRAPH | GRAPH[S] ( * | name[, ...] )) [
  ELEMENT[S] ( * | label-or-rel-type[, ...] )
  | NODE[S] ( * | label[, ...] )
  | RELATIONSHIP[S] ( * | rel-type[, ...] )
] TO role[, ...]

For example, to grant the role regularUsers the ability to SET any property on all elements of the graph neo4j, use:

GRANT SET PROPERTY { } ON HOME GRAPH ELEMENTS * TO regularUsers

The SET PROPERTY privilege can also be denied:

DENY SET PROPERTY "{" ( * | property[, ...] ) "}" ON (HOME GRAPH | GRAPH[S] ( * | name[, ...] )) [
  ELEMENT[S] ( * | label-or-rel-type[, ...] )
  | NODE[S] ( * | label[, ...] )
  | RELATIONSHIP[S] ( * | rel-type[, ...] )
] TO role[, ...]

For example, to deny the role regularUsers the ability to SET the property foo on nodes with the label bar on all graphs, use:

DENY SET PROPERTY { foo } ON GRAPH * NODES bar TO regularUsers

If the user attempts to set a property with a property name that does not already exist on the database, the user must also possess the CREATE NEW PROPERTY NAME privilege.

27.7.6. The MERGE privilege

The MERGE privilege is a compound privilege that combines TRAVERSE and READ (i.e. MATCH) with CREATE and SET PROPERTY. This is intended to enable the use of the MERGE command, but it is also applicable to all reads and writes that require these privileges.

GRANT MERGE "{" ( * | property[, ...] ) "}" ON (HOME GRAPH | GRAPH[S] ( * | name[, ...] )) [
  ELEMENT[S] ( * | label-or-rel-type[, ...] )
  | NODE[S] ( * | label[, ...] )
  | RELATIONSHIP[S] ( * | rel-type[, ...] )
] TO role[, ...]

For example, to grant the role regularUsers the ability to MERGE on all elements of the graph neo4j, use:

GRANT MERGE { } ON GRAPH neo4j ELEMENTS * TO regularUsers
It is not possible to deny the MERGE privilege. If you wish to prevent a user from creating elements and setting properties: use DENY CREATE or DENY SET PROPERTY.

If the user attempts to create nodes with a label that does not already exist on the database, the user must also possess the CREATE NEW LABEL privilege. The same applies to new relationships and properties - the CREATE NEW RELATIONSHIP TYPE or CREATE NEW PROPERTY NAME privileges are required.

27.7.7. The WRITE privilege

The WRITE privilege allows the user to execute any WRITE command on a graph.

```plaintext
GRANT WRITE
   ON (HOME GRAPH | GRAPH[S] { * | name[, ...] })
   TO role[, ...]
```

For example, to grant the role regularUsers the ability to WRITE on the graph neo4j, use:

```plaintext
GRANT WRITE ON GRAPH neo4j TO regularUsers
```

Unlike the more specific WRITE commands, it is not possible to restrict WRITE privileges to specific ELEMENTS, NODES or RELATIONSHIPS. If you wish to prevent a user from writing to a subset of database objects, a GRANT WRITE can be combined with more specific DENY commands to target these elements.

The WRITE privilege can also be denied:

```plaintext
DENY WRITE
   ON (HOME GRAPH | GRAPH[S] { * | name[, ...] })
   TO role[, ...]
```

For example, to deny the role regularUsers the ability to WRITE on the graph neo4j, use:

```plaintext
DENY WRITE ON GRAPH neo4j TO regularUsers
```

Users with WRITE privilege but restricted TRAVERSE privileges will not be able to do DETACH DELETE in all cases. See Operations Manual → Fine-grained access control for more info.

27.7.8. The ALL GRAPH PRIVILEGES privilege

The ALL GRAPH PRIVILEGES privilege allows the user to execute any command on a graph:

```plaintext
GRANT ALL [[ GRAPH ] PRIVILEGES ]
   ON (HOME GRAPH | GRAPH[S] { * | name[, ...] })
   TO role[, ...]
```

For example, to grant the role regularUsers ALL GRAPH PRIVILEGES on the graph neo4j, use:
GRANT ALL GRAPH PRIVILEGES ON GRAPH neo4j TO regularUsers

Unlike the more specific READ and WRITE commands, it is not possible to restrict ALL GRAPH PRIVILEGES to specific ELEMENTS, NODES or RELATIONSHIPS. If you wish to prevent a user from reading or writing to a subset of database objects, a GRANT ALL GRAPH PRIVILEGES can be combined with more specific DENY commands to target these elements.

The ALL GRAPH PRIVILEGES privilege can also be denied:

DENY ALL [ [ GRAPH ] PRIVILEGES ]
    ON { HOME GRAPH | GRAPH[S] { * | name[, ...] } }
    TO role[, ...]

For example, to deny the role regularUsers all graph privileges on the graph neo4j, use:

DENY ALL GRAPH PRIVILEGES ON GRAPH neo4j TO regularUsers

27.8. Database administration

This section explains how to use Cypher to manage Neo4j database administrative privileges.

Administrators can use the following Cypher commands to manage Neo4j database administrative rights. The components of the database privilege commands are:

- the commands:
  - GRANT – gives privileges to roles.
  - DENY – denies privileges to roles.
  - REVOKE – removes granted or denied privileges from roles.

- database-privilege
  - ACCESS - allows access to a specific database or remote database alias.
  - START - allows the specified database to be started.
  - STOP - allows the specified database to be stopped.
  - CREATE INDEX - allows indexes to be created on the specified database.
  - DROP INDEX - allows indexes to be deleted on the specified database.
  - SHOW INDEX - allows indexes to be listed on the specified database.
  - INDEX [MANAGEMENT] - allows indexes to be created, deleted, and listed on the specified database.
  - CREATE CONSTRAINT - allows constraints to be created on the specified database.
  - DROP CONSTRAINT - allows constraints to be deleted on the specified database.
° **SHOW CONSTRAINT** - allows constraints to be listed on the specified database.

° **CONSTRAINT [MANAGEMENT]** - allows constraints to be created, deleted, and listed on the specified database.

° **CREATE NEW [NODE] LABEL** - allows labels to be created so that future nodes can be assigned them.

° **CREATE NEW [RELATIONSHIP] TYPE** - allows relationship types to be created, so that relationships can be assigned to them.

° **CREATE NEW [PROPERTY] NAME** - allows property names to be created, so that nodes and relationships can have properties assigned with these names.

° **NAME [MANAGEMENT]** - allows all of the name management capabilities: node labels, relationship types, and property names.

° **ALL [[DATABASE] PRIVILEGES]** - allows access, index, constraint, and name management for the specified database or remote database alias.

° **SHOW TRANSACTION** - allows listing transactions and queries for the specified users on the specified database.

° **TERMINATE TRANSACTION** - allows ending transactions and queries for the specified users on the specified database.

° **TRANSACTION [MANAGEMENT]** - allows listing and ending transactions and queries for the specified users on the specified database.

• **name**

° The database to associate the privilege with.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANT database-privilege ON {HOME DATABASE</td>
<td>DATABASE[S] {*</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>DENY</strong> database-privilege ON (HOME DATABASE</td>
<td>DATABASE[S] {*</td>
</tr>
<tr>
<td><strong>REVOKE</strong> GRANT database-privilege ON (HOME DATABASE</td>
<td>DATABASE[S] {*</td>
</tr>
<tr>
<td><strong>REVOKE</strong> DENY database-privilege ON (HOME DATABASE</td>
<td>DATABASE[S] {*</td>
</tr>
<tr>
<td><strong>REVOKE</strong> database-privilege ON (HOME DATABASE</td>
<td>DATABASE[S] {*</td>
</tr>
</tbody>
</table>

**DENY** does NOT erase a granted privilege. Use **REVOKE** if you want to remove a privilege.

The hierarchy between the different database privileges is shown in the image below.

![Database privileges hierarchy](image_url)

**Figure 4. Database privileges hierarchy**

**Table 531. Database privilege command syntax**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
</table>
| **GRANT** ACCESS ON (HOME DATABASE | DATABASE[S] {* | name[, ...]]) TO role[, ...] | Grants the specified roles the privilege to access:

- the home database
- specific database(s) or remote database alias(es)
- all databases and remote database aliases |
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANT {START</td>
<td>STOP} \n  ON (HOME DATABASE</td>
</tr>
<tr>
<td>GRANT {CREATE</td>
<td>DROP</td>
</tr>
<tr>
<td>GRANT INDEX[ES] [MANAGEMENT] \n  ON (HOME DATABASE</td>
<td>DATABASES {*</td>
</tr>
<tr>
<td>GRANT {CREATE</td>
<td>DROP</td>
</tr>
<tr>
<td>GRANT CONSTRAINT[S] [MANAGEMENT] \n  ON (HOME DATABASE</td>
<td>DATABASES {*</td>
</tr>
<tr>
<td>GRANT CREATE NEW {NODE } LABEL[S] \n  ON (HOME DATABASE</td>
<td>DATABASES {*</td>
</tr>
<tr>
<td>GRANT CREATE NEW {RELATIONSHIP } TYPE[S] \n  ON (HOME DATABASE</td>
<td>DATABASES {*</td>
</tr>
<tr>
<td>GRANT CREATE NEW {PROPERTY } NAME[S] \n  ON (HOME DATABASE</td>
<td>DATABASES {*</td>
</tr>
</tbody>
</table>
27.8.1. The database **ACCESS** privilege

The **ACCESS** privilege enables users to connect to a database or a remote database alias. With **ACCESS** you can run calculations, for example, `RETURN 2*5 AS answer` or call functions `RETURN timestamp() AS time`. 
GRANT ACCESS
  ON {HOME DATABASE | DATABASE[S] {*, | name[,...]}}
TO role[,...]

For example, to grant the role regularUsers the ability to access the database neo4j, use:

GRANT ACCESS ON DATABASE neo4j TO regularUsers

The ACCESS privilege can also be denied:

DENY ACCESS
  ON {HOME DATABASE | DATABASE[S] {*, | name[,...]}}
TO role[,...]

For example, to deny the role regularUsers the ability to access the remote database alias remote-db, use:

DENY ACCESS ON DATABASE `remote-db` TO regularUsers

The privileges granted can be seen using the SHOW PRIVILEGES command:

SHOW ROLE regularUsers PRIVILEGES AS COMMANDS

Table 532. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;DENY ACCESS ON DATABASE remote-db TO <code>regularUsers</code>&quot;</td>
</tr>
<tr>
<td>&quot;GRANT ACCESS ON DATABASE neo4j TO <code>regularUsers</code>&quot;</td>
</tr>
</tbody>
</table>

Rows: 2

27.8.2. The database START/STOP privileges

The START privilege can be used to enable the ability to start a database:

GRANT START
  ON {HOME DATABASE | DATABASE[S] {*, | name[,...]}}
TO role[,...]

For example, to grant the role regularUsers the ability to start the database neo4j, use:

GRANT START ON DATABASE neo4j TO regularUsers

The START privilege can also be denied:
For example, to deny the role `regularUsers` the ability to start the database `neo4j`, use:

```
DENY START ON DATABASE system TO regularUsers
```

The `STOP` privilege can be used to enable the ability to stop a database:

```
GRANT STOP ON DATABASE neo4j TO regularUsers
```

The `STOP` privilege can also be denied:

```
DENY STOP ON DATABASE system TO regularUsers
```

For example, to deny the role `regularUsers` the ability to stop the database `neo4j`, use:

```
DENY STOP ON DATABASE system TO regularUsers
```

The privileges granted can be seen using the `SHOW PRIVILEGES` command:

```
SHOW ROLE regularUsers PRIVILEGES AS COMMANDS
```

Table 533. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;DENY ACCESS ON DATABASE remote-db TO <code>regularUsers</code>&quot;</td>
</tr>
<tr>
<td>&quot;DENY START ON DATABASE system TO <code>regularUsers</code>&quot;</td>
</tr>
<tr>
<td>&quot;DENY STOP ON DATABASE system TO <code>regularUsers</code>&quot;</td>
</tr>
<tr>
<td>&quot;GRANT ACCESS ON DATABASE neo4j TO <code>regularUsers</code>&quot;</td>
</tr>
<tr>
<td>&quot;GRANT START ON DATABASE neo4j TO <code>regularUsers</code>&quot;</td>
</tr>
<tr>
<td>&quot;GRANT STOP ON DATABASE neo4j TO <code>regularUsers</code>&quot;</td>
</tr>
</tbody>
</table>

Rows: 6

Note that `START` and `STOP` privileges are not included in the `ALL DATABASE PRIVILEGES`. 
27.8.3. The **INDEX MANAGEMENT** privileges

Indexes can be created, deleted, or listed with the `CREATE INDEX`, `DROP INDEX`, and `SHOW INDEXES` commands. The privilege to do this can be granted with `GRANT CREATE INDEX`, `GRANT DROP INDEX`, and `GRANT SHOW INDEX` commands. The privilege to do all three can be granted with `GRANT INDEX MANAGEMENT` command.

Table 534. Index management command syntax

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANT {CREATE</td>
<td>DROP</td>
</tr>
<tr>
<td>GRANT INDEX[ES] [MANAGEMENT] ON {HOME DATABASE</td>
<td>DATABASE[ES] {*</td>
</tr>
</tbody>
</table>

For example, to grant the role `regularUsers` the ability to create indexes on the database `neo4j`, use:

```
GRANT CREATE INDEX ON DATABASE neo4j TO regularUsers
```

The `SHOW INDEXES` privilege only affects the `SHOW INDEXES` command, and not the older procedures for listing indexes, such as `db.indexes`.

27.8.4. The **CONSTRAINT MANAGEMENT** privileges

Constraints can be created, deleted, or listed with the `CREATE CONSTRAINT`, `DROP CONSTRAINT` and `SHOW CONSTRAINTS` commands. The privilege to do this can be granted with `GRANT CREATE CONSTRAINT`, `GRANT DROP CONSTRAINT`, and `GRANT SHOW CONSTRAINT` commands. The privilege to do all three can be granted with `GRANT CONSTRAINT MANAGEMENT` command.

Table 535. Constraint management command syntax

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANT {CREATE</td>
<td>DROP</td>
</tr>
<tr>
<td>GRANT CONSTRAINT[ES] [MANAGEMENT] ON {HOME DATABASE</td>
<td>DATABASE[ES] {*</td>
</tr>
</tbody>
</table>

For example, to grant the role `regularUsers` the ability to create constraints on the database `neo4j`, use:

```
GRANT CREATE CONSTRAINT ON DATABASE neo4j TO regularUsers
```
GRANT CREATE CONSTRAINT ON DATABASE neo4j TO regularUsers

The SHOW CONSTRAINTS privilege only affects the SHOW CONSTRAINTS command, and not the older procedures for listing constraints, such as `db.constraints`.

27.8.5. The **NAME MANAGEMENT** privileges

The right to create new labels, relationship types, and property names is different from the right to create nodes, relationships, and properties. The latter is managed using database WRITE privileges, while the former is managed using specific GRANT/DENY CREATE NEW ... commands for each type.

Table 536. Label, relationship type and property name management command syntax

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANT CREATE NEW [NODE] LABEL[S]</td>
<td>Enables the specified roles to create new node labels in the home database, specific database(s), or all databases.</td>
</tr>
<tr>
<td>ON (HOME DATABASE</td>
<td>DATABASE[S] [{*</td>
</tr>
<tr>
<td>GRANT CREATE NEW [RELATIONSHIP] TYPE[S]</td>
<td>Enables the specified roles to create new relationship types in the home database, specific database(s), or all databases.</td>
</tr>
<tr>
<td>ON (HOME DATABASE</td>
<td>DATABASE[S] [{*</td>
</tr>
<tr>
<td>GRANT CREATE NEW [PROPERTY] NAME[S]</td>
<td>Enables the specified roles to create new property names in the home database, specific database(s), or all databases.</td>
</tr>
<tr>
<td>ON (HOME DATABASE</td>
<td>DATABASE[S] [{*</td>
</tr>
<tr>
<td>GRANT NAME [MANAGEMENT]</td>
<td>Enables the specified roles to create new labels, relationship types, and property names in the home database, specific database(s), or all databases.</td>
</tr>
<tr>
<td>ON (HOME DATABASE</td>
<td>DATABASE[S] [{*</td>
</tr>
</tbody>
</table>

For example, to grant the role `regularUsers` the ability to create new properties on nodes or relationships on the database `neo4j`, use:

GRANT CREATE NEW PROPERTY NAME ON DATABASE neo4j TO regularUsers

27.8.6. Granting **ALL DATABASE PRIVILEGES**

The right to access a database, create and drop indexes and constraints and create new labels, relationship types or property names can be achieved with a single command:

GRANT ALL [[DATABASE] PRIVILEGES]                                       |
| ON (HOME DATABASE | DATABASE[S] [{* | name[ , ...]}]) TO role[ , ...]                              |
Note that the privileges for starting and stopping all databases, and transaction management, are not included in the **ALL DATABASE PRIVILEGES** grant. These privileges are associated with administrators while other database privileges are of use to domain and application developers.

For example, granting the abilities above on the database `neo4j` to the role `databaseAdminUsers` is done using the following query.

```sql
GRANT ALL DATABASE PRIVILEGES ON DATABASE neo4j TO databaseAdminUsers
```

The privileges granted can be seen using the **SHOW PRIVILEGES** command:

```sql
SHOW ROLE databaseAdminUsers PRIVILEGES AS COMMANDS
```

### Table 537. Result

<table>
<thead>
<tr>
<th>command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT ALL DATABASE PRIVILEGES ON DATABASE neo4j TO 'databaseAdminUsers'&quot;</td>
<td>Enables the specified roles to list transactions and queries for user(s) or all users in the home database, specific database(s), or all databases.</td>
</tr>
</tbody>
</table>

#### 27.8.7. Granting TRANSACTION MANAGEMENT privileges

The right to run the commands **SHOW TRANSACTIONS**, **TERMINATE TRANSACTIONS**, and the deprecated procedures `dbms.listTransactions`, `dbms.listQueries`, `dbms.killQuery`, `dbms.killQueries`, `dbms.killTransaction` and `dbms.killTransactions` is now managed through the **SHOW TRANSACTION** and **TERMINATE TRANSACTION** privileges.

### Table 538. Transaction management command syntax

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANT SHOW TRANSACTION[S] (( {*</td>
<td>user[, ...] } )) ON {HOME DATABASE</td>
</tr>
</tbody>
</table>

| GRANT TERMINATE TRANSACTION[S] (( {* | user[, ...] } )) ON {HOME DATABASE | DATABASE[S] {* | name[, ...]}} TO role[, ...] | Enables the specified roles to end running transactions and queries for user(s) or all users in the home database, specific database(s), or all databases. |

| GRANT TRANSACTION [MANAGEMENT] (( {* | user[, ...] } )) ON {HOME DATABASE | DATABASE[S] {* | name[, ...]}} TO role[, ...] | Enables the specified roles to manage transactions and queries for user(s) or all users in the home database, specific database(s), or all databases. |

Note that the TRANSACTION MANAGEMENT privileges are not included in the **ALL DATABASE PRIVILEGES**.
For example, to grant the role `regularUsers` the ability to list transactions for user `jake` on the database `neo4j`, use:

```
GRANT SHOW TRANSACTION (jake) ON DATABASE neo4j TO regularUsers
```

27.9. DBMS administration

This section explains how to use Cypher to manage Neo4j DBMS administrative privileges.

All DBMS privileges are relevant system-wide. Like user management, they do not belong to one specific database or graph. For more details on the differences between graphs, databases and the DBMS, refer to Neo4j databases and graphs.

![Figure 6. Syntax of GRANT and DENY DBMS Privileges](image)

![Figure 7. DBMS privileges hierarchy](image)

The `admin` role has a number of built-in privileges. These include:
• Create, delete, and modify databases and aliases.
• Change configuration parameters.
• Manage transactions.
• Manage users and roles.
• Manage sub-graph privileges.
• Manage procedure security.

To enable a user to perform these tasks, you can grant them the admin role, but it is also possible to make a custom role with a subset of these privileges. All privileges are also assignable using Cypher commands. For more details, see the following sections:

• Role management
• User management
• Impersonation privileges management
• Database management
• Alias management
• Privilege management
• Transaction management
• Procedure and user defined function security

27.9.1. Using a custom role to manage DBMS privileges

In order to have an administrator role with a subset of privileges that includes all DBMS privileges, but not all database privileges, you can copy the admin role and revoke or deny the unwanted privileges. A second option is to build a custom administrator from scratch by granting the wanted privileges instead.

As an example, an administrator role can be created to only manage users and roles by using the second option:

1. First, create the new role:

   CREATE ROLE usermanager

2. Then grant the privilege to manage users:

   GRANT USER MANAGEMENT ON DBMS TO usermanager

3. And to manage roles:

   GRANT ROLE MANAGEMENT ON DBMS TO usermanager

The resulting role has privileges that only allow user and role management. To list all privileges for the role usermanager as commands, run this query:
Table 539. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT ROLE MANAGEMENT ON DBMS TO 'usermanager'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT USER MANAGEMENT ON DBMS TO 'usermanager'&quot;</td>
</tr>
</tbody>
</table>

Rows: 2

Note that this role doesn’t allow all DBMS capabilities. For example, the role is missing privileges for management, creation and drop of databases as well as execution of admin procedures. To create a more powerful administrator, you can grant a different set of privileges.

In the following example, a new administrator role is created to perform almost all DBMS capabilities, excluding database management. However, the role still has some limited database capabilities, such as managing transactions:

1. Again, start by creating a new role:

   CREATE ROLE customAdministrator

2. Then grant the privilege for all DBMS capabilities:

   GRANT ALL DBMS PRIVILEGES ON DBMS TO customAdministrator

3. And explicitly deny the privilege to manage databases and aliases:

   DENY DATABASE MANAGEMENT ON DBMS TO customAdministrator

4. Next, grant the transaction management privilege:

   GRANT TRANSACTION MANAGEMENT (*) ON DATABASE * TO customAdministrator

The resulting role has privileges that include all DBMS privileges except creating, dropping, and modifying databases and aliases, as well as managing transactions. Use the following query to list all privileges for the role customAdministrator as commands:

SHOW ROLE customAdministrator PRIVILEGES AS COMMANDS

Table 540. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;DENY DATABASE MANAGEMENT ON DBMS TO 'customAdministrator'&quot;</td>
</tr>
<tr>
<td>&quot;GRANT ALL DBMS PRIVILEGES ON DBMS TO 'customAdministrator'&quot;</td>
</tr>
</tbody>
</table>
27.9.2. The DBMS **ROLE MANAGEMENT** privileges

The DBMS privileges for role management are assignable using Cypher administrative commands. They can be granted, denied and revoked like other privileges.

Table 541. Role management privileges command syntax

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANT CREATE ROLE ON DBMS TO role[, ...]</td>
<td>Enables the specified roles to create new roles.</td>
</tr>
<tr>
<td>GRANT RENAME ROLE ON DBMS TO role[, ...]</td>
<td>Enables the specified roles to change the name of roles.</td>
</tr>
<tr>
<td>GRANT DROP ROLE ON DBMS TO role[, ...]</td>
<td>Enables the specified roles to delete roles.</td>
</tr>
<tr>
<td>GRANT ASSIGN ROLE ON DBMS TO role[, ...]</td>
<td>Enables the specified roles to assign roles to users.</td>
</tr>
<tr>
<td>GRANT REMOVE ROLE ON DBMS TO role[, ...]</td>
<td>Enables the specified roles to remove roles from users.</td>
</tr>
<tr>
<td>GRANT SHOW ROLE ON DBMS TO role[, ...]</td>
<td>Enables the specified roles to list roles.</td>
</tr>
<tr>
<td>GRANT ROLE MANAGEMENT ON DBMS TO role[, ...]</td>
<td>Enables the specified roles to create, delete, assign, remove, and list roles.</td>
</tr>
</tbody>
</table>

The ability to add roles can be granted via the **CREATE ROLE** privilege. See an example:

```
GRANT CREATE ROLE ON DBMS TO roleAdder
```
The resulting role has privileges that only allow adding roles. List all privileges for the role `roleAdder` as commands by using the following query:

```
SHOW ROLE roleAdder PRIVILEGES AS COMMANDS
```

Table 542. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT CREATE ROLE ON DBMS TO 'roleAdder'&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

The ability to rename roles can be granted via the `RENAME ROLE` privilege. See an example:

```
GRANT RENAME ROLE ON DBMS TO roleNameModifier
```

The resulting role has privileges that only allow renaming roles. List all privileges for the role `roleNameModifier` using the following query:

```
SHOW ROLE roleNameModifier PRIVILEGES AS COMMANDS
```

Table 543. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT RENAME ROLE ON DBMS TO 'roleNameModifier'&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

The ability to delete roles can be granted via the `DROP ROLE` privilege. See an example:

```
GRANT DROP ROLE ON DBMS TO roleDropper
```

The resulting role has privileges that only allow deleting roles. List all privileges for the role `roleDropper` by using the following query:

```
SHOW ROLE roleDropper PRIVILEGES AS COMMANDS
```

Table 544. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT DROP ROLE ON DBMS TO 'roleDropper'&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

The ability to assign roles to users can be granted via the `ASSIGN ROLE` privilege. See an example:
The resulting role has privileges that only allow assigning/granting roles. List all privileges for the role `roleAssigner` as commands by using the following query:

```
SHOW ROLE roleAssigner PRIVILEGES AS COMMANDS
```

Table 545. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT ASSIGN ROLE ON DBMS TO 'roleAssigner'&quot;</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

The ability to remove roles from users can be granted via the `REMOVE ROLE` privilege. See an example:

```
GRANT REMOVE ROLE ON DBMS TO roleRemover
```

The resulting role has privileges that only allow removing/revoking roles. List all privileges for the role `roleRemover` as commands by using the following query:

```
SHOW ROLE roleRemover PRIVILEGES AS COMMANDS
```

Table 546. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT REMOVE ROLE ON DBMS TO 'roleRemover'&quot;</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

The ability to show roles can be granted via the `SHOW ROLE` privilege. A user with this privilege is allowed to execute the `SHOW ROLES` and `SHOW POPULATED ROLES` administration commands.

In order to use `SHOW ROLES WITH USERS` and `SHOW POPULATED ROLES WITH USERS` administration commands, both this and the `SHOW USER` privileges are required. See an example of how to grant the `SHOW ROLE` privilege:

```
GRANT SHOW ROLE ON DBMS TO roleShower
```

The resulting role has privileges that only allow showing roles. List all privileges for the role `roleShower` as commands by using the following query:

```
SHOW ROLE roleShower PRIVILEGES AS COMMANDS
```

Table 547. Result
The privileges to create, rename, delete, assign, remove, and list roles can be granted via the ROLE MANAGEMENT privilege. See an example:

```
GRANT ROLE MANAGEMENT ON DBMS TO roleManager
```

The resulting role has all privileges to manage roles. List all privileges for the role `roleManager` as commands by using the following query:

```
SHOW ROLE roleManager PRIVILEGES AS COMMANDS
```

Table 548. Result

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GRANT ROLE MANAGEMENT ON DBMS TO 'roleManager'</code></td>
<td>Enables the specified roles to create new users.</td>
</tr>
<tr>
<td><code>GRANT RENAME USER ON DBMS TO role[,...]</code></td>
<td>Enables the specified roles to change the name of users.</td>
</tr>
<tr>
<td><code>GRANT ALTER USER ON DBMS TO role[,...]</code></td>
<td>Enables the specified roles to modify users.</td>
</tr>
</tbody>
</table>

27.9.3. The DBMS USER MANAGEMENT privileges

The DBMS privileges for user management can be assigned using Cypher administrative commands. They can be granted, denied and revoked like other privileges.

Table 549. User management privileges command syntax

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GRANT CREATE USER ON DBMS TO role[,...]</code></td>
<td>Enables the specified roles to create new users.</td>
</tr>
<tr>
<td><code>GRANT RENAME USER ON DBMS TO role[,...]</code></td>
<td>Enables the specified roles to change the name of users.</td>
</tr>
<tr>
<td><code>GRANT ALTER USER ON DBMS TO role[,...]</code></td>
<td>Enables the specified roles to modify users.</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>GRANT SET PASSWORD[S] ON DBMS TO role[, ...]</td>
<td>Enables the specified roles to modify users' passwords and whether those passwords must be changed upon first login.</td>
</tr>
<tr>
<td>GRANT SET USER HOME DATABASE ON DBMS TO role[, ...]</td>
<td>Enables the specified roles to modify users' home database.</td>
</tr>
<tr>
<td>GRANT SET USER STATUS ON DBMS TO role[, ...]</td>
<td>Enables the specified roles to modify the account status of users.</td>
</tr>
<tr>
<td>GRANT DROP USER ON DBMS TO role[, ...]</td>
<td>Enables the specified roles to delete users.</td>
</tr>
<tr>
<td>GRANT SHOW USER ON DBMS TO role[, ...]</td>
<td>Enables the specified roles to list users.</td>
</tr>
<tr>
<td>GRANT USER MANAGEMENT ON DBMS TO role[, ...]</td>
<td>Enables the specified roles to create, delete, modify, and list users.</td>
</tr>
</tbody>
</table>

The ability to add users can be granted via the `CREATE USER` privilege. See an example:

```
GRANT CREATE USER ON DBMS TO userAdder
```

The resulting role has privileges that only allow adding users. List all privileges for the role `userAdder` as commands by using this query:

```
SHOW ROLE userAdder PRIVILEGES AS COMMANDS
```

<table>
<thead>
<tr>
<th>Table 550. Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
</tr>
</tbody>
</table>
| "GRANT CREATE USER ON DBMS TO `userAdder`"

Rows: 1

The ability to rename users can be granted via the `RENAME USER` privilege. The following query shows an example of this:
GRANT RENAME USER ON DBMS TO `userNameModifier`

The resulting role has privileges that only allow renaming users:

SHOW ROLE `userNameModifier` PRIVILEGES AS COMMANDS

Lists all privileges for role `userNameModifier`:

Table 551. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT RENAME USER ON DBMS TO <code>userNameModifier</code>&quot;</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

The ability to modify users can be granted via the ALTER USER privilege. See an example:

GRANT ALTER USER ON DBMS TO `userModifier`

The resulting role has privileges that only allow modifying users. List all privileges for the role `userModifier` as commands by using the following query:

SHOW ROLE `userModifier` PRIVILEGES AS COMMANDS

Table 552. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT ALTER USER ON DBMS TO <code>userModifier</code>&quot;</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

A user that is granted the ALTER USER privilege is allowed to run the ALTER USER administration command with one or several of the SET PASSWORD, SET PASSWORD CHANGE [NOT] REQUIRED and SET STATUS parts:

ALTER USER `jake` SET PASSWORD 'secret' SET STATUS SUSPENDED

The ability to modify users' passwords and whether those passwords must be changed upon first login can be granted via the SET PASSWORDS privilege. See an example:

GRANT SET PASSWORDS ON DBMS TO `passwordModifier`

The resulting role has privileges that only allow modifying users' passwords and whether those passwords must be changed upon first login. List all privileges for the role `passwordModifier` as commands by using the following query:
A user that is granted the `SET PASSWORDS` privilege is allowed to run the `ALTER USER` administration command with one or both of the `SET PASSWORD` and `SET PASSWORD CHANGE [NOT] REQUIRED` parts:

```
ALTER USER jake SET PASSWORD 'abc123' CHANGE NOT REQUIRED
```

The ability to modify the account status of users can be granted via the `SET USER STATUS` privilege. See an example:

```
GRANT SET USER STATUS ON DBMS TO statusModifier
```

The resulting role has privileges that only allow modifying the account status of users. List all privileges for the role `statusModifier` as commands by using the following query:

```
SHOW ROLE statusModifier PRIVILEGES AS COMMANDS
```

A user that is granted the `SET USER STATUS` privilege is allowed to run the `ALTER USER` administration command with only the `SET STATUS` part:

```
ALTER USER jake SET STATUS ACTIVE
```

In order to be able to modify the home database of users, grant the `SET USER HOME DATABASE` privilege. See an example:

```
GRANT SET USER HOME DATABASE ON DBMS TO statusModifier
```

The resulting role has privileges that only allow modifying the home database of users. List all privileges for the role `statusModifier` as commands by using the following query:

```
SHOW ROLE statusModifier PRIVILEGES AS COMMANDS
```
A user that is granted the `SET USER HOME DATABASE` privilege is allowed to run the `ALTER USER` administration command with only the `SET HOME DATABASE` or `REMOVE HOME DATABASE` part:

```
ALTER USER jake SET HOME DATABASE otherDb
```

```
ALTER USER jake REMOVE HOME DATABASE
```

Note that the combination of the `SET PASSWORDS`, `SET USER STATUS`, and the `SET USER HOME DATABASE` privilege actions is equivalent to the `ALTER USER` privilege action.

The ability to delete users can be granted via the `DROP USER` privilege. See an example:

```
GRANT DROP USER ON DBMS TO userDropper
```

The resulting role has privileges that only allow deleting users. List all privileges for the role `userDropper` as commands by using the following query:

```
SHOW ROLE userDropper PRIVILEGES AS COMMANDS
```

The ability to show users can be granted via the `SHOW USER` privilege. See an example:

```
GRANT SHOW USER ON DBMS TO userShower
```

The resulting role has privileges that only allow showing users. List all privileges for the role `userShower` as commands by using the following query:

```
SHOW ROLE userShower PRIVILEGES AS COMMANDS
```

---

<table>
<thead>
<tr>
<th>Table 555. Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>command</strong></td>
</tr>
<tr>
<td>&quot;GRANT SET USER HOME DATABASE ON DBMS TO <code>statusModifier</code>&quot;</td>
</tr>
<tr>
<td>&quot;GRANT SET USER STATUS ON DBMS TO <code>statusModifier</code>&quot;</td>
</tr>
<tr>
<td>Rows: 2</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Table 556. Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>command</strong></td>
</tr>
<tr>
<td>&quot;GRANT DROP USER ON DBMS TO <code>userDropper</code>&quot;</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Table 557. Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>command</strong></td>
</tr>
<tr>
<td>&quot;GRANT SHOW USER ON DBMS TO <code>userShower</code>&quot;</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>
The privileges to create, rename, modify, delete, and list users can be granted via the `USER MANAGEMENT` privilege. See an example:

```
GRANT USER MANAGEMENT ON DBMS TO userManager
```

The resulting role has all privileges to manage users. List all privileges for the role `userManager` as commands by using the following query:

```
SHOW ROLE userManager PRIVILEGES AS COMMANDS
```

### Table 558. Result

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&quot;GRANT SHOW USER ON DBMS TO '&quot;userShower&quot;'&quot;</code></td>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

#### 27.9.4. The DBMS `IMPERSONATE` privileges

The DBMS privileges for impersonation can be assigned through Cypher administrative commands. They can be granted, denied, and revoked like other privileges.

Impersonation is the ability of a user to assume another user’s roles (and therefore privileges), with the restriction of not being able to execute updating `admin` commands as the impersonated user (i.e. they would still be able to use `SHOW` commands).

The ability to impersonate users can be granted via the `IMPERSONATE` privilege.

### Table 559. Impersonation privileges command syntax

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
</table>
| `GRANT IMPERSONATE [(*)]`  
  `ON DBMS`  
  `TO role[, ...]` | Enables the specified roles to impersonate any user. |
| `GRANT IMPERSONATE (user[, ...])`  
  `ON DBMS`  
  `TO role[, ...]` | Enables the specified roles to impersonate the specified users. |

The following query shows an example of this. Note that `userImpersonator` must be an existing role in order to make this query work:
Query

GRANT IMPERSONATE (*) ON DBMS TO userImpersonator

The resulting role has privileges that allow impersonating all users:

Query

SHOW ROLE userImpersonator PRIVILEGES AS COMMANDS

Table 560. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT IMPERSONATE (*) ON DBMS TO <code>userImpersonator</code>&quot;</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

It is also possible to deny and revoke that privilege. See an example which shows how the userImpersonator user would be able to impersonate all users, except alice:

Query

DENY IMPERSONATE (alice) ON DBMS TO userImpersonator

To grant (or revoke) the permissions to impersonate a specific user or a subset of users, you can first list them with this query:

Query

GRANT IMPERSONATE (alice, bob) ON DBMS TO userImpersonator

27.9.5. The DBMS DATABASE MANAGEMENT privileges

The DBMS privileges for database management can be assigned by using Cypher administrative commands. They can be granted, denied and revoked like other privileges.

Table 561. Database management privileges command syntax

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANT CREATE DATABASE ON DBMS TO role[,...]</td>
<td>Enables the specified roles to create new databases and aliases.</td>
</tr>
<tr>
<td>GRANT DROP DATABASE ON DBMS TO role[,...]</td>
<td>Enables the specified roles to delete databases and aliases.</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>GRANT ALTER DATABASE ON DBMS TO role[,...]</td>
<td>Enables the specified roles to modify databases and aliases.</td>
</tr>
<tr>
<td>GRANT SET DATABASE ACCESS ON DBMS TO role[,...]</td>
<td>Enables the specified roles to modify access of databases.</td>
</tr>
<tr>
<td>GRANT DATABASE MANAGEMENT ON DBMS TO role[,...]</td>
<td>Enables the specified roles to create, delete, and modify databases and aliases.</td>
</tr>
</tbody>
</table>

The ability to create databases and aliases can be granted via the `CREATE DATABASE` privilege. See an example:

```
GRANT CREATE DATABASE ON DBMS TO databaseAdder
```

The resulting role has privileges that only allow creating databases and aliases. List all privileges for the role `databaseAdder` as commands by using the following query:

```
SHOW ROLE databaseAdder PRIVILEGES AS COMMANDS
```

Table 562. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT CREATE DATABASE ON DBMS TO <code>databaseAdder</code>&quot;</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

The ability to delete databases and aliases can be granted via the `DROP DATABASE` privilege. See an example:

```
GRANT DROP DATABASE ON DBMS TO databaseDropper
```

The resulting role has privileges that only allow deleting databases and aliases. List all privileges for the role `databaseDropper` as commands by using the following query:

```
SHOW ROLE databaseDropper PRIVILEGES AS COMMANDS
```

Table 563. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT DROP DATABASE ON DBMS TO <code>databaseDropper</code>&quot;</td>
</tr>
</tbody>
</table>
The ability to modify databases and aliases can be granted via the `ALTER DATABASE` privilege. See an example:

```
GRANT ALTER DATABASE ON DBMS TO databaseModifier
```

The resulting role has privileges that only allow modifying databases and aliases. List all privileges for the role `databaseModifier` as commands by using the following query:

```
SHOW ROLE databaseModifier PRIVILEGES AS COMMANDS
```

Table 564. Result

```
command
"GRANT ALTER DATABASE ON DBMS TO `databaseModifier`"
Rows: 1
```

The ability to modify access of databases can be granted via the `SET DATABASE ACCESS` privilege. See an example:

```
GRANT SET DATABASE ACCESS ON DBMS TO accessModifier
```

The resulting role has privileges that only allow modifying access of databases. List all privileges for the role `accessModifier` as commands by using the following query:

```
SHOW ROLE accessModifier PRIVILEGES AS COMMANDS
```

Table 565. Result

```
command
"GRANT SET DATABASE ACCESS ON DBMS TO `accessModifier`"
Rows: 1
```

The privileges to create, delete, and modify databases and aliases can be granted via the `DATABASE MANAGEMENT` privilege. See an example:

```
GRANT DATABASE MANAGEMENT ON DBMS TO databaseManager
```

The resulting role has all privileges to manage databases and aliases. List all privileges for the role `databaseManager` as commands by using the following query:
SHOW ROLE databaseManager PRIVILEGES AS COMMANDS

Table 566. Result

<table>
<thead>
<tr>
<th>command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT DATABASE MANAGEMENT ON DBMS TO <code>databaseManager</code>&quot;</td>
<td></td>
</tr>
</tbody>
</table>

Rows: 1

27.9.6. The DBMS ** Alias MANAGEMENT ** privileges

The DBMS privileges for alias management can be assigned by using Cypher administrative commands and can be applied to both local and remote aliases. They can be granted, denied and revoked like other privileges. It is also possible to manage aliases with [database management commands](#).

Table 567. Alias management privileges command syntax

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANT CREATE ALIAS ON DBMS TO role[,...]</td>
<td>Enables the specified roles to create new aliases.</td>
</tr>
<tr>
<td>GRANT DROP ALIAS ON DBMS TO role[,...]</td>
<td>Enables the specified roles to delete aliases.</td>
</tr>
<tr>
<td>GRANT ALTER ALIAS ON DBMS TO role[,...]</td>
<td>Enables the specified roles to modify aliases.</td>
</tr>
<tr>
<td>GRANT SHOW ALIAS ON DBMS TO role[,...]</td>
<td>Enables the specified roles to list aliases.</td>
</tr>
<tr>
<td>GRANT ALIAS MANAGEMENT ON DBMS TO role[,...]</td>
<td>Enables the specified roles to list, create, delete, and modify aliases.</td>
</tr>
</tbody>
</table>

The ability to create aliases can be granted via the **CREATE ALIAS** privilege. See an example:

GRANT CREATE ALIAS ON DBMS TO aliasAdder

The resulting role has privileges that only allow creating aliases. List all privileges for the role aliasAdder as commands by using the following query:
SHOW ROLE aliasAdder PRIVILEGES AS COMMANDS

Table 568. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT CREATE ALIAS ON DBMS TO <code>aliasAdder</code>&quot;</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

The ability to delete aliases can be granted via the DROP ALIAS privilege. See an example:

GRANT DROP ALIAS ON DBMS TO aliasDropper

The resulting role has privileges that only allow deleting aliases. See all privileges for the role aliasDropper as commands by using the following query:

SHOW ROLE aliasDropper PRIVILEGES AS COMMANDS

Table 569. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT DROP ALIAS ON DBMS TO <code>aliasDropper</code>&quot;</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

The ability to modify aliases can be granted via the ALTER ALIAS privilege. See an example:

GRANT ALTER ALIAS ON DBMS TO aliasModifier

The resulting role has privileges that only allow modifying aliases. List all privileges for the role aliasModifier as commands by using the following query:

SHOW ROLE aliasModifier PRIVILEGES AS COMMANDS

Table 570. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT ALTER ALIAS ON DBMS TO <code>aliasModifier</code>&quot;</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

The ability to list aliases can be granted via the SHOW ALIAS privilege. See an example:

GRANT SHOW ALIAS ON DBMS TO aliasLister

The resulting role has privileges that only allow modifying aliases. List all privileges for the role
aliasLister as commands by using the following query:

```
SHOW ROLE aliasLister PRIVILEGES AS COMMANDS
```

Table 571. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT SHOW ALIAS ON DBMS TO 'aliasLister'&quot;</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

The privileges to list, create, delete, and modify aliases can be granted via the ALIAS MANAGEMENT privilege. See an example:

```
GRANT ALIAS MANAGEMENT ON DBMS TO aliasManager
```

The resulting role has all privileges to manage aliases. List all privileges for the role aliasManager as commands by using the following query:

```
SHOW ROLE aliasManager PRIVILEGES AS COMMANDS
```

Table 572. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT ALIAS MANAGEMENT ON DBMS TO 'aliasManager'&quot;</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

27.9.7. The DBMS PRIVILEGE MANAGEMENT privileges

The DBMS privileges for privilege management can be assigned by using Cypher administrative commands. They can be granted, denied and revoked like other privileges.

Table 573. Privilege management privileges command syntax

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANT SHOW PRIVILEGE ON DBMS TO role[, ...]</td>
<td>Enables the specified roles to list privileges.</td>
</tr>
<tr>
<td>GRANT ASSIGN PRIVILEGE ON DBMS TO role[, ...]</td>
<td>Enables the specified roles to assign privileges using the GRANT and DENY commands.</td>
</tr>
</tbody>
</table>
### Command | Description
--- | ---
GRANT REMOVE PRIVILEGE ON DBMS TO role[, ...] | Enables the specified roles to remove privileges using the REVOKE command.
GRANT PRIVILEGE MANAGEMENT ON DBMS TO role[, ...] | Enables the specified roles to list, assign, and remove privileges.

The ability to list privileges can be granted via the SHOW PRIVILEGE privilege. A user with this privilege is allowed to execute the SHOW PRIVILEGES and SHOW ROLE roleName PRIVILEGES administration commands. To execute the SHOW USER username PRIVILEGES administration command, both this privilege and the SHOW USER privilege are required. The following query shows an example of how to grant the SHOW PRIVILEGE privilege:

```
grant show privilege on dbms to `privilegeShower`
```

The resulting role has privileges that only allow showing privileges. List all privileges for the role `privilegeShower` as commands by using the following query:

```
show role `privilegeShower` privileges as commands
```

Table 574. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT SHOW PRIVILEGE ON DBMS TO <code>privilegeShower</code>&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

Note that no specific privileges are required for showing the current user's privileges through the SHOW USER username PRIVILEGES or SHOW USER PRIVILEGES commands.

In addition, note that if a non-native auth provider like LDAP is in use, SHOW USER PRIVILEGES will only work with a limited capacity by making it only possible for a user to show their own privileges. Other users' privileges cannot be listed when using a non-native auth provider.

The ability to assign privileges to roles can be granted via the ASSIGN PRIVILEGE privilege. A user with this privilege is allowed to execute GRANT and DENY administration commands. See an example of how to grant this privilege:

```
grant assign privilege on dbms to `privilegeAssigner`
```

The resulting role has privileges that only allow assigning privileges. List all privileges for the role `privilegeAssigner` as commands by using the following query:
The ability to remove privileges from roles can be granted via the `REMOVE PRIVILEGE` privilege. A user with this privilege is allowed to execute `REVOKE` administration commands. See an example of how to grant this privilege:

```sql
GRANT REMOVE PRIVILEGE ON DBMS TO `privilegeRemover`
```

The resulting role has privileges that only allow removing privileges. List all privileges for the role `privilegeRemover` as commands by using the following query:

```sql
SHOW ROLE privilegeRemover PRIVILEGES AS COMMANDS
```

Table 576. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT REMOVE PRIVILEGE ON DBMS TO <code>privilegeRemover</code>&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

The privileges to list, assign, and remove privileges can be granted via the `PRIVILEGE MANAGEMENT` privilege. See an example:

```sql
GRANT PRIVILEGE MANAGEMENT ON DBMS TO `privilegeManager`
```

The resulting role has all privileges to manage privileges. List all privileges for the role `privilegeManager` as commands by using the following query:

```sql
SHOW ROLE privilegeManager PRIVILEGES AS COMMANDS
```

Table 577. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT PRIVILEGE MANAGEMENT ON DBMS TO <code>privilegeManager</code>&quot;</td>
</tr>
</tbody>
</table>

Rows: 1
27.9.8. The DBMS EXECUTE privileges

The DBMS privileges for procedure and user defined function execution can be assigned by using Cypher administrative commands. They can be granted, denied and revoked like other privileges.

Table 578. Execute privileges command syntax

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANT EXECUTE PROCEDURE[S] name-globbing[, ...] ON DBMS TO role[, ...]</td>
<td>Enables the specified roles to execute the given procedures.</td>
</tr>
<tr>
<td>GRANT EXECUTE BOOSTED PROCEDURE[S] name-globbing[, ...] ON DBMS TO role[, ...]</td>
<td>Enables the specified roles to execute the given procedures with elevated privileges.</td>
</tr>
<tr>
<td>GRANT EXECUTE ADMINISTRATOR PROCEDURES ON DBMS TO role[, ...]</td>
<td>Enables the specified roles to execute procedures annotated with @Admin. The procedures are executed with elevated privileges.</td>
</tr>
<tr>
<td>GRANT EXECUTE [USER [DEFINED]] FUNCTION[S] name-globbing[, ...] ON DBMS TO role[, ...]</td>
<td>Enables the specified roles to execute the given user defined functions.</td>
</tr>
<tr>
<td>GRANT EXECUTE BOOSTED [USER [DEFINED]] FUNCTION[S] name-globbing[, ...] ON DBMS TO role[, ...]</td>
<td>Enables the specified roles to execute the given user defined functions with elevated privileges.</td>
</tr>
</tbody>
</table>

The EXECUTE BOOSTED privileges replace the dbms.security.procedures.default_allowed and dbms.security.procedures.roles configuration parameters for procedures and user defined functions. The configuration parameters are still honored as a set of temporary privileges. These cannot be revoked, but will be updated on each restart with the current configuration values.

The EXECUTE PROCEDURE privilege

The ability to execute a procedure can be granted via the EXECUTE PROCEDURE privilege. A user with this privilege is allowed to execute the procedures matched by the name-globbing. The following query shows an example of how to grant this privilege:

```
GRANT EXECUTE PROCEDURE db.schema.* ON DBMS TO procedureExecutor
```

Users with the role procedureExecutor can then run any procedure in the db.schema namespace. The procedure is run using the user's own privileges.

The resulting role has privileges that only allow executing procedures in the db.schema namespace. List all
privileges for the role `procedureExecutor` as commands by using the following query:

```sql
SHOW ROLE procedureExecutor PRIVILEGES AS COMMANDS
```

<table>
<thead>
<tr>
<th>Table 579. Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
</tr>
<tr>
<td>&quot;GRANT EXECUTE PROCEDURE db.schema.* ON DBMS TO <code>procedureExecutor</code>&quot;</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

In order to allow the execution of all but only a few procedures, you can grant `EXECUTE PROCEDURES *` and deny the unwanted procedures. For example, the following queries allow the execution of all procedures, except those starting with `dbms.killTransaction`:

```sql
GRANT EXECUTE PROCEDURE * ON DBMS TO deniedProcedureExecutor

DENY EXECUTE PROCEDURE dbms.killTransaction* ON DBMS TO deniedProcedureExecutor
```

The resulting role has privileges that only allow executing all procedures except those starting with `dbms.killTransaction`. List all privileges for the role `deniedProcedureExecutor` as commands by using the following query:

```sql
SHOW ROLE deniedProcedureExecutor PRIVILEGES AS COMMANDS
```

<table>
<thead>
<tr>
<th>Table 580. Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
</tr>
</tbody>
</table>
| "DENY EXECUTE PROCEDURE dbms.killTransaction* ON DBMS TO `deniedProcedureExecutor`"
| "GRANT EXECUTE PROCEDURE * ON DBMS TO `deniedProcedureExecutor`"
| Rows: 2          |

Both `dbms.killTransaction` and `dbms.killTransactions` are blocked here, as well as any other procedures starting with `dbms.killTransaction`.

The **EXECUTE BOOSTED PROCEDURE** privilege

The ability to execute a procedure with elevated privileges can be granted via the `EXECUTE BOOSTED PROCEDURE` privilege. A user with this privilege is allowed to execute the procedures matched by the name-globbing without the execution being restricted to their other privileges.

There is no need to grant an individual `EXECUTE PROCEDURE` privilege for the procedures either, as granting the `EXECUTE BOOSTED PROCEDURE` includes an implicit `EXECUTE PROCEDURE` grant for them. A denied `EXECUTE PROCEDURE` still denies executing the procedure. The following query shows an example of how to grant this privilege:
Users with the role `boostedProcedureExecutor` can thus run `db.labels` and `db.relationshipTypes` with full privileges. Now they can see everything on the graph and not just the labels and types that the user has `TRAVERSE` privilege on.

The resulting role has privileges that only allow executing procedures `db.labels` and `db.relationshipTypes`, but with elevated execution. List all privileges for the role `boostedProcedureExecutor` as commands by using the following query:

```
SHOW ROLE `boostedProcedureExecutor` PRIVILEGES AS COMMANDS
```

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT EXECUTE BOOSTED PROCEDURE db.labels ON DBMS TO <code>boostedProcedureExecutor</code>&quot;</td>
</tr>
<tr>
<td>&quot;GRANT EXECUTE BOOSTED PROCEDURE db.relationshipTypes ON DBMS TO <code>boostedProcedureExecutor</code>&quot;</td>
</tr>
</tbody>
</table>

Rows: 2

Granting the `EXECUTE BOOSTED PROCEDURE` privilege on its own allows the procedure to be both executed (due to the implicit `EXECUTE PROCEDURE` grant) and proceed with elevated privileges. A denied `EXECUTE BOOSTED PROCEDURE` on its own behaves slightly differently: it only denies the elevation and not the execution of the procedure. However, a role with both a granted `EXECUTE BOOSTED PROCEDURE` and a denied `EXECUTE BOOSTED PROCEDURE` will deny the execution as well. This is explained through the following examples:
Example 16. Grant `EXECUTE PROCEDURE` and deny `EXECUTE BOOSTED PROCEDURE`

```
GRANT EXECUTE PROCEDURE * ON DBMS TO `deniedBoostedProcedureExecutor1`

DENY EXECUTE BOOSTED PROCEDURE db.labels ON DBMS TO `deniedBoostedProcedureExecutor1`
```

The resulting role has privileges that allow the execution of all procedures using the user’s own privileges. It also prevents `db.labels` from being elevated. Still, the denied `EXECUTE BOOSTED PROCEDURE` does not block execution of `db.labels`.

To list all privileges for role `deniedBoostedProcedureExecutor1` as commands, use the following query:

```
SHOW ROLE `deniedBoostedProcedureExecutor1` PRIVILEGES AS COMMANDS
```

Table 582. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;DENY EXECUTE BOOSTED PROCEDURE db.labels ON DBMS TO <code>deniedBoostedProcedureExecutor1</code>&quot;</td>
</tr>
<tr>
<td>&quot;GRANT EXECUTE PROCEDURE * ON DBMS TO <code>deniedBoostedProcedureExecutor1</code>&quot;</td>
</tr>
</tbody>
</table>

Rows: 2

Example 17. Grant `EXECUTE BOOSTED PROCEDURE` and deny `EXECUTE PROCEDURE`

```
GRANT EXECUTE BOOSTED PROCEDURE * ON DBMS TO `deniedBoostedProcedureExecutor2`

DENY EXECUTE PROCEDURE db.labels ON DBMS TO `deniedBoostedProcedureExecutor2`
```

The resulting role has privileges that allow executing all procedures with elevated privileges except `db.labels`, which is not allowed to be executed at all. List all privileges for the role `deniedBoostedProcedureExecutor2` as commands by using the following query:

```
SHOW ROLE `deniedBoostedProcedureExecutor2` PRIVILEGES AS COMMANDS
```

Table 583. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;DENY EXECUTE PROCEDURE db.labels ON DBMS TO <code>deniedBoostedProcedureExecutor2</code>&quot;</td>
</tr>
<tr>
<td>&quot;GRANT EXECUTE BOOSTED PROCEDURE * ON DBMS TO <code>deniedBoostedProcedureExecutor2</code>&quot;</td>
</tr>
</tbody>
</table>

Rows: 2
**Example 18. Grant** EXECUTE BOOSTED PROCEDURE and deny EXECUTE BOOSTED PROCEDURE

```
GRANT EXECUTE BOOSTED PROCEDURE * ON DBMS TO `deniedBoostedProcedureExecutor3`

DENY EXECUTE BOOSTED PROCEDURE db.labels ON DBMS TO `deniedBoostedProcedureExecutor3`
```

The resulting role has privileges that allow executing all procedures with elevated privileges except `db.labels`, which is not allowed to be executed at all. List all privileges for the role `deniedBoostedProcedureExecutor3` as commands by using the following query:

```
SHOW ROLE `deniedBoostedProcedureExecutor3` PRIVILEGES AS COMMANDS
```

**Table 584. Result**

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;DENY EXECUTE BOOSTED PROCEDURE db.labels ON DBMS TO <code>deniedBoostedProcedureExecutor3</code>&quot;</td>
</tr>
<tr>
<td>&quot;GRANT EXECUTE BOOSTED PROCEDURE * ON DBMS TO <code>deniedBoostedProcedureExecutor3</code>&quot;</td>
</tr>
</tbody>
</table>

Rows: 2
Example 19. Grant `EXECUTE PROCEDURE` and `EXECUTE BOOSTED PROCEDURE` and deny `EXECUTE BOOSTED PROCEDURE`

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANT EXECUTE PROCEDURE db.labels ON DBMS TO <code>deniedBoostedProcedureExecutor4</code></td>
</tr>
<tr>
<td>GRANT EXECUTE BOOSTED PROCEDURE * ON DBMS TO <code>deniedBoostedProcedureExecutor4</code></td>
</tr>
<tr>
<td>DENY EXECUTE BOOSTED PROCEDURE db.labels ON DBMS TO <code>deniedBoostedProcedureExecutor4</code></td>
</tr>
</tbody>
</table>

The resulting role has privileges that allow executing all procedures with elevated privileges except `db.labels`, which is only allowed to execute using the user’s own privileges. List all privileges for the role `deniedBoostedProcedureExecutor4` as commands by using the following query:

```sql
SHOW ROLE deniedBoostedProcedureExecutor4 PRIVILEGES AS COMMANDS
```

<table>
<thead>
<tr>
<th>Table 585. Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
</tr>
<tr>
<td>&quot;DENY EXECUTE BOOSTED PROCEDURE db.labels ON DBMS TO <code>deniedBoostedProcedureExecutor4</code>&quot;</td>
</tr>
<tr>
<td>&quot;GRANT EXECUTE BOOSTED PROCEDURE * ON DBMS TO <code>deniedBoostedProcedureExecutor4</code>&quot;</td>
</tr>
<tr>
<td>&quot;GRANT EXECUTE PROCEDURE db.labels ON DBMS TO <code>deniedBoostedProcedureExecutor4</code>&quot;</td>
</tr>
</tbody>
</table>

Rows: 3
Example 20. How would the privileges from examples 1 to 4 affect the output of a procedure?

Assume there is a procedure called `myProc`.

This procedure gives the result `A` and `B` for a user with `EXECUTE PROCEDURE` privilege and `A`, `B` and `C` for a user with `EXECUTE BOOSTED PROCEDURE` privilege.

Now, adapt the privileges from examples 1 to 4 to be applied to this procedure and show what is returned. With the privileges from example 1, granted `EXECUTE PROCEDURE *` and denied `EXECUTE BOOSTED PROCEDURE myProc`, the `myProc` procedure returns the result `A` and `B`.

With the privileges from example 2, granted `EXECUTE BOOSTED PROCEDURE *` and denied `EXECUTE PROCEDURE myProc`, execution of the `myProc` procedure is not allowed.

With the privileges from example 3, granted `EXECUTE BOOSTED PROCEDURE *` and denied `EXECUTE BOOSTED PROCEDURE myProc`, execution of the `myProc` procedure is not allowed.

With the privileges from example 4, granted `EXECUTE PROCEDURE myProc` and `EXECUTE BOOSTED PROCEDURE *` and denied `EXECUTE BOOSTED PROCEDURE myProc`, the `myProc` procedure returns the result `A` and `B`.

For comparison, when only `EXECUTE BOOSTED PROCEDURE myProc` is granted, the `myProc` procedure returns the result `A`, `B` and `C`, without the need for granting of the `EXECUTE PROCEDURE myProc` privilege.

The `EXECUTE ADMIN PROCEDURE` privilege

The ability to execute admin procedures (annotated with `@Admin`) can be granted via the `EXECUTE ADMIN PROCEDURES` privilege. This privilege is equivalent to granting the `EXECUTE BOOSTED PROCEDURE` privilege on each of the admin procedures. Any newly added admin procedure is automatically included in this privilege. The following query shows an example of how to grant this privilege:

```
GRANT EXECUTE ADMIN PROCEDURES ON DBMS TO adminProcedureExecutor
```

Users with the role `adminProcedureExecutor` can then run any admin procedure with elevated privileges.

The resulting role has privileges that allow executing all admin procedures. List all privileges for the role `adminProcedureExecutor` as commands by using the following query:

```
SHOW ROLE adminProcedureExecutor PRIVILEGES AS COMMANDS
```

<table>
<thead>
<tr>
<th>Table 586. Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
</tr>
<tr>
<td>&quot;GRANT EXECUTE ADMIN PROCEDURES ON DBMS TO <code>adminProcedureExecutor</code>&quot;</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

In order to compare this with the `EXECUTE PROCEDURE` and `EXECUTE BOOSTED PROCEDURE` privileges, revisit the
myProc procedure, but this time as an admin procedure, which will give the result A, B and C when allowed to execute.

By starting with a user only granted with the EXECUTE PROCEDURE myProc privilege, execution of the myProc procedure is not allowed.

However, for a user granted with the EXECUTE BOOSTED PROCEDURE myProc or EXECUTE ADMIN PROCEDURES privileges, the myProc procedure returns the result A, B and C.

Any denied EXECUTE privilege results in the procedure not being allowed to be executed. In this case, it does not matter whether EXECUTE PROCEDURE, EXECUTE BOOSTED PROCEDURE or EXECUTE ADMIN PROCEDURES is being denied.

The EXECUTE USER DEFINED FUNCTION privilege

The ability to execute a user defined function (UDF) can be granted via the EXECUTE USER DEFINED FUNCTION privilege. A user with this privilege is allowed to execute the UDFs matched by the name-globbing.

The EXECUTE USER DEFINED FUNCTION privilege does not apply to built-in functions, which are always executable.

Example 21. Execute user defined function

The following query shows an example of how to grant this privilege:

```
GRANT EXECUTE USER DEFINED FUNCTION apoc.coll.* ON DBMS TO functionExecutor
```

Or in short form:

```
GRANT EXECUTE FUNCTION apoc.coll.* ON DBMS TO functionExecutor
```

Users with the role functionExecutor can thus run any UDF in the apoc.coll namespace. The function here is run using the user's own privileges.

The resulting role has privileges that only allow executing UDFs in the apoc.coll namespace. List all privileges for the role functionExecutor as commands by using the following query:

```
SHOW ROLE functionExecutor PRIVILEGES AS COMMANDS
```

<table>
<thead>
<tr>
<th>Table 587. Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
</tr>
<tr>
<td>&quot;GRANT EXECUTE FUNCTION apoc.coll.* ON DBMS TO 'functionExecutor'&quot;</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>
To allow the execution of all but a few UDFs, you can grant `EXECUTE USER DEFINED FUNCTIONS *` and deny the unwanted functions.

Example 22. Execute user defined functions

The following queries allow the execution of all UDFs except those starting with `apoc.any.prop`:

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANT EXECUTE USER DEFINED FUNCTIONS * ON DBMS TO <code>deniedFunctionExecutor</code></td>
</tr>
<tr>
<td>DENY EXECUTE USER DEFINED FUNCTION <code>apoc.any.prop*</code> ON DBMS TO <code>deniedFunctionExecutor</code></td>
</tr>
</tbody>
</table>

Or in short form:

- GRANT EXECUTE FUNCTIONS * ON DBMS TO `deniedFunctionExecutor`
- DENY EXECUTE FUNCTION `apoc.any.prop*` ON DBMS TO `deniedFunctionExecutor`

The resulting role has privileges that only allow the execution of all procedures except those starting with `apoc.any.prop`. List all privileges for the role `deniedFunctionExecutor` as commands by using the following query:

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHOW ROLE <code>deniedFunctionExecutor</code> PRIVILEGES AS COMMANDS</td>
</tr>
</tbody>
</table>

Table 588. Result

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;DENY EXECUTE FUNCTION <code>apoc.any.prop*</code> ON DBMS TO <code>deniedFunctionExecutor</code>&quot;</td>
</tr>
<tr>
<td>&quot;GRANT EXECUTE FUNCTION * ON DBMS TO <code>deniedFunctionExecutor</code>&quot;</td>
</tr>
</tbody>
</table>

Rows: 2

The `apoc.any.property` and `apoc.any.properties` are blocked, as well as any other procedures starting with `apoc.any.prop`.

The `EXECUTE BOOSTED USER DEFINED FUNCTION` privilege

The ability to execute a user defined function (UDF) with elevated privileges can be granted via the `EXECUTE BOOSTED USER DEFINED FUNCTION` privilege. A user with this privilege is allowed to execute the UDFs matched by the name-globbing without the execution being restricted to their other privileges.

There is no need to grant an individual `EXECUTE USER DEFINED FUNCTION` privilege for the functions, as granting `EXECUTE BOOSTED USER DEFINED FUNCTION` includes an implicit `EXECUTE USER DEFINED FUNCTION` grant. However, a denied `EXECUTE USER DEFINED FUNCTION` still prevents the function to be executed.
The `EXECUTE BOOSTED USER DEFINED FUNCTION` privilege does not apply to built-in functions, as they have no concept of elevated privileges.

Granting `EXECUTE BOOSTED USER DEFINED FUNCTION` on its own allows the UDF to be both executed (because of the implicit `EXECUTE USER DEFINED FUNCTION` grant) and gives it elevated privileges during the execution. A denied `EXECUTE BOOSTED USER DEFINED FUNCTION` on its own behaves slightly differently: it only denies the elevation and not the execution of the UDF. However, a role with only a granted `EXECUTE BOOSTED USER DEFINED FUNCTION` and a denied `EXECUTE BOOSTED USER DEFINED FUNCTION` prevents the execution to be performed as well. This is the same behavior as for the `EXECUTE BOOSTED PROCEDURE` privilege.

Example 23. Execute boosted user defined function

The following query shows an example of how to grant the `EXECUTE BOOSTED USER DEFINED FUNCTION` privilege:

```
GRANT EXECUTE BOOSTED USER DEFINED FUNCTION apoc.any.properties ON DBMS TO boostedFunctionExecutor
```

Or in short form:

```
GRANT EXECUTE BOOSTED FUNCTION apoc.any.properties ON DBMS TO boostedFunctionExecutor
```

Users with the role `boostedFunctionExecutor` can thus run `apoc.any.properties` with full privileges and see every property on the node/relationship, not just the properties that the user has `READ` privilege on.

The resulting role has privileges that only allow executing of the UDF `apoc.any.properties`, but with elevated execution. List all privileges for the role `boostedFunctionExecutor` as commands by using the following query:

```
SHOW ROLE boostedFunctionExecutor PRIVILEGES AS COMMANDS
```

Table 589. Result

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT EXECUTE BOOSTED FUNCTION apoc.any.properties ON DBMS TO 'boostedFunctionExecutor'&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

Procedure and user-defined function name-globbing

The name-globbing for procedure and user defined function names is a simplified version of globbing for filename expansions. It only allows two wildcard characters: `*` and `?`, which are used for multiple and single character matches. In this case, `*` means 0 or more characters and `?` matches exactly one character.
The name-globbing is subject to the standard Cypher restrictions on valid identifiers, with the exception that it may include dots, stars, and question marks without the need for escaping using backticks.

Each part of the name-globbing separated by dots may be individually escaped, for example, `mine.` procedureWith%` but not `mine.procedure`With%`. It is also good to keep in mind that wildcard characters behave as wildcards even when escaped. As an example, using `*` is equivalent to using *, and thus allows executing all functions or procedures and not only the procedure or function named *.

The examples below only use procedures, but the same rules apply to user defined function names:

- `mine.public.exampleProcedure`
- `mine.public.exampleProcedure1`
- `mine.public.exampleProcedure2`
- `mine.public.with#Special§Characters`
- `mine.private.exampleProcedure`
- `mine.private.exampleProcedure1`
- `mine.private.exampleProcedure2`
- `mine.private.with#Special§Characters`
- `your.exampleProcedure`

```
GRANT EXECUTE PROCEDURE * ON DBMS TO globbing1
```

Users with the role `globbing1` can thus run all the procedures.

```
GRANT EXECUTE PROCEDURE mine.*.exampleProcedure ON DBMS TO globbing2
```

Users with the role `globbing2` can thus run procedures `mine.public.exampleProcedure` and `mine.private.exampleProcedure`, but none of the others.

```
GRANT EXECUTE PROCEDURE mine.*.exampleProcedure? ON DBMS TO globbing3
```

Users with the role `globbing3` can thus run procedures `mine.public.exampleProcedure1`, `mine.private.exampleProcedure1` and `mine.private.exampleProcedure2`, but none of the others.

```
GRANT EXECUTE PROCEDURE *.exampleProcedure ON DBMS TO globbing4
```

Users with the role `globbing4` can thus run procedures `your.exampleProcedure`, `mine.public.exampleProcedure`, and `mine.private.exampleProcedure`, but none of the others.

```
GRANT EXECUTE PROCEDURE mine.public.exampleProcedure* ON DBMS TO globbing5
```
27.9.9. Granting **ALL DBMS PRIVILEGES**

The right to perform the following privileges can be achieved with a single command:

- Create, drop, assign, remove and show roles
- Create, alter, drop, show and impersonate users
- Create, alter and drop databases
- Show, assign and remove privileges
- Execute all procedures with elevated privileges
- Execute all user defined functions with elevated privileges

```
GRANT ALL DBMS PRIVILEGES ON DBMS TO role [, ...]
```

For example, to grant the role `dbmsManager` the abilities above, use the following query:

```
GRANT ALL DBMS PRIVILEGES ON DBMS TO dbmsManager
```

The privileges granted can be seen using the `SHOW PRIVILEGES` command:

```
SHOW ROLE dbmsManager PRIVILEGES AS COMMANDS
```

**Table 590. Result**

<table>
<thead>
<tr>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANT ALL DBMS PRIVILEGES ON DBMS TO <code>dbmsManager</code>&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

27.10. Limitations

*This section lists the known limitations and implications of Neo4js role-based access*
27.10.1. Security and Indexes

As described in Indexes for search performance, Neo4j 4.4 supports the creation and use of indexes to improve the performance of Cypher queries.

Note that the Neo4j security model impacts the results of queries, regardless if the indexes are used or not. When using non full-text Neo4j indexes, a Cypher query will always return the same results it would have if no index existed. This means that, if the security model causes fewer results to be returned due to restricted read access in Graph and sub-graph access control, the index will also return the same fewer results.

However, this rule is not fully obeyed by Indexes for full-text search. These specific indexes are backed by Lucene internally. It is therefore not possible to know for certain whether a security violation has affected each specific entry returned from the index. In face of this, Neo4j will return zero results from full-text indexes in case it is determined that any result might be violating the security privileges active for that query.

Since full-text indexes are not automatically used by Cypher, they do not lead to the case where the same Cypher query would return different results simply because such an index was created. Users need to explicitly call procedures to use these indexes. The problem is only that, if this behavior is not known by the user, they might expect the full-text index to return the same results that a different, but semantically similar, Cypher query does.

Example with denied properties

Consider the following example. The database has nodes with labels :User and :Person, and they have properties name and surname. There are indexes on both properties:

```
CREATE INDEX singleProp FOR (n:User) ON (n.name);
CREATE INDEX composite FOR (n:User) ON (n.name, n.surname);
CREATE FULLTEXT INDEX userNames FOR (n:User|Person) ON EACH [n.name, n.surname];
```

Full-text indexes support multiple labels. See Indexes for full-text search for more details on creating and using full-text indexes.

After creating these indexes, it would appear that the latter two indexes accomplish the same thing. However, this is not completely accurate. The composite and full-text indexes behave in different ways and are focused on different use cases. A key difference is that full-text indexes are backed by Lucene, and will use the Lucene syntax for querying.

This has consequences for users restricted on the labels or properties involved in the indexes. Ideally, if the labels and properties in the index are denied, they can correctly return zero results from both native indexes and full-text indexes. However, there are borderline cases where this is not as simple.

Imagine the following nodes were added to the database:
CREATE (:User {name: 'Sandy'});
CREATE (:User {name: 'Mark', surname: 'Andy'});
CREATE (:User {name: 'Andy', surname: 'Anderson'});
CREATE (:User:Person {name: 'Mandy', surname: 'Smith'});
CREATE (:User:Person {name: 'Joe', surname: 'Andy'});

Consider denying the label :Person:

DENY TRAVERSE Person ON GRAPH * TO users;

If the user runs a query that uses the native single property index on name:

MATCH (n:User) WHERE n.name CONTAINS 'ndy' RETURN n.name;

This query performs several checks:

- Scans the index to create a stream of results of nodes with the name property, which leads to five results.
- Filters the results to include only nodes where n.name CONTAINS 'ndy', filtering out Mark and Joe, which leads to three results.
- Filters the results to exclude nodes that also have the denied label :Person, filtering out Mandy, which leads to two results.

Two results will be returned from this dataset and only one of them has the surname property.

In order to use the native composite index on name and surname, the query needs to include a predicate on the surname property as well:

MATCH (n:User) WHERE n.name CONTAINS 'ndy' AND n.surname IS NOT NULL RETURN n.name;

This query performs several checks, which are almost identical to the single property index query:

- Scans the index to create a stream of results of nodes with the name and surname property, which leads to four results.
- Filters the results to include only nodes where n.name CONTAINS 'ndy', filtering out Mark and Joe, which leads to two results.
- Filters the results to exclude nodes that also have the denied label :Person, filtering out Mandy, which leads to only one result.

Only one result was returned from the above dataset. What if this query with the full-text index was used instead:

CALL db.index.fulltext.queryNodes("userNames", "ndy") YIELD node, score RETURN node.name

The problem now is that it is not certain whether the results provided by the index were achieved due to a match to the name or the surname property. The steps taken by the query engine would be:
• Run a Lucene query on the full-text index to produce results containing \textit{ndy} in either property, leading to five results.

• Filter the results to exclude nodes that also have the label \texttt{:Person}, filtering out \textit{Mandy} and \textit{Joe}, leading to three results.

This difference in results is caused by the \texttt{OR} relationship between the two properties in the index creation.

Denying properties

Now consider denying access on properties, like the \texttt{surname} property:

\begin{verbatim}
DENY READ {surname} ON GRAPH * TO users;
\end{verbatim}

For that, run the same queries again:

\begin{verbatim}
MATCH (n:User) WHERE n.name CONTAINS 'ndy' RETURN n.name;
\end{verbatim}

This query operates exactly as before, returning the same two results, because nothing in it relates to the denied property.

However, this is not the same for the query targeting the composite index:

\begin{verbatim}
MATCH (n:User) WHERE n.name CONTAINS 'ndy' AND n.surname IS NOT NULL RETURN n.name;
\end{verbatim}

Since the \texttt{surname} property is denied, it will appear to always be \texttt{null} and the composite index empty. Therefore, the query returns no result.

Now consider the full-text index query:

\begin{verbatim}
CALL db.index.fulltext.queryNodes("userNames", "ndy") YIELD node, score
RETURN node.name
\end{verbatim}

The problem remains, since it is not certain whether the results provided by the index were returned due to a match on the \texttt{name} or the \texttt{surname} property. Results from the \texttt{surname} property now need to be excluded by the security rules, because they require that the user is unable to see any \texttt{surname} properties. However, the security model is not able to introspect the Lucene query in order to know what it will actually do, whether it works only on the allowed \texttt{name} property, or also on the disallowed \texttt{surname} property. What is known is that the earlier query returned a match for \textit{Joe Andy} which should now be filtered out. Therefore, in order to never return results the user should not be able to see, all results need to be blocked. The steps taken by the query engine would be:

• Determine if the full-text index includes denied properties.

• If yes, return an empty results stream. Otherwise, it will process as described before.

In this case, the query will return zero results rather than simply returning the results \textit{Andy} and \textit{Sandy}, which might have been expected.
27.10.2. Security and labels

Traversing the graph with multi-labeled nodes

The general influence of access control privileges on graph traversal is described in detail in Graph and sub-graph access control. The following section will only focus on nodes due to their ability to have multiple labels. Relationships can only have one type of label and thus they do not exhibit the behavior this section aims to clarify. While this section will not mention relationships further, the general function of the traverse privilege also applies to them.

For any node that is traversable, due to `GRANT TRAVERSE` or `GRANT MATCH`, the user can get information about the attached labels by calling the built-in `labels()` function. In the case of nodes with multiple labels, they can be returned to users that weren’t directly granted access to.

To give an illustrative example, imagine a graph with three nodes: one labeled :A, another labeled :B and one with the labels :A and :B. In this case, there is a user with the role `custom` defined by:

```
GRANT TRAVERSE ON GRAPH * NODES A TO custom;
```

If that user were to execute

```
MATCH (n:A) RETURN n, labels(n);
```

They would get a result with two nodes: the node that was labeled with :A and the node with labels :A :B.

In contrast, executing

```
MATCH (n:B) RETURN n, labels(n);
```

This will return only the one node that has both labels: :A and :B. Even though :B did not have access to traversals, there is one node with that label accessible in the dataset due to the allow-listed label :A that is attached to the same node.

If a user is denied to traverse on a label they will never get results from any node that has this label attached to it. Thus, the label name will never show up for them. As an example, this can be done by executing:

```
DENY TRAVERSE ON GRAPH * NODES B TO custom;
```

The query

```
MATCH (n:A) RETURN n, labels(n);
```

will now return the node only labeled with :A, while the query

```
MATCH (n:B) RETURN n, labels(n);
```
will now return no nodes.

The db.labels() procedure

In contrast to the normal graph traversal described in the previous section, the built-in `db.labels()` procedure is not processing the data graph itself, but the security rules defined on the system graph. That means:

- If a label is explicitly whitelisted (granted), it will be returned by this procedure.
- If a label is denied or isn’t explicitly allowed, it will not be returned by this procedure.

Reusing the previous example, imagine a graph with three nodes: one labeled :A, another labeled :B and one with the labels :A and :B. In this case, there is a user with the role `custom` defined by:

```plaintext
GRANT TRAVERSE ON GRAPH * NODES A TO custom;
```

This means that only label :A is explicitly allow-listed. Thus, executing

```plaintext
CALL db.labels();
```

will only return label :A, because that is the only label for which traversal was granted.

27.10.3. Security and count store operations

The rules of a security model may impact some of the database operations. This means extra security checks are necessary to incur additional data accesses, especially in the case of count store operations. These are, however, usually very fast lookups and the difference might be noticeable.

See the following security rules that set up a `restricted` and a `free` role as an example:

```plaintext
GRANT TRAVERSE ON GRAPH * NODES Person TO restricted;
DENY TRAVERSE ON GRAPH * NODES Customer TO restricted;
GRANT TRAVERSE ON GRAPH * ELEMENTS * TO free;
```

Now, let’s look at what the database needs to do in order to execute the following query:

```plaintext
MATCH (n:Person) RETURN count(n);
```

For both roles the execution plan will look like this:

```
+--------------------------+
| Operator                |
| +ProduceResults         |
| |                        |
| +NodeCountFromCountStore |
```

Internally, however, very different operations need to be executed. The following table illustrates the difference:
<table>
<thead>
<tr>
<th>User with <strong>free role</strong></th>
<th>User with <strong>restricted role</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>The database can access the count store and retrieve the total number of nodes with the label <code>:Person</code>.</td>
<td>The database cannot access the count store because it must make sure that only traversable nodes with the desired label <code>:Person</code> are counted. Due to this, each node with the <code>:Person</code> label needs to be accessed and examined to make sure that they do not have a deny-listed label, such as <code>:Customer</code>. Due to the additional data accesses that the security checks need to do, this operation will be slower compared to executing the query as an unrestricted user.</td>
</tr>
<tr>
<td>This is a very quick operation.</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 28. Query tuning

This section describes query tuning for the Cypher query language.

Neo4j aims 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 the very least, data should get filtered out as early as possible in order to reduce the amount of work that has to be done in the later stages of query execution. This also applies to what gets returned: returning whole nodes and relationships ought to be avoided in favour of selecting and returning only the data that is needed. 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 query planner. To minimize the resources used for this, try 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
  - Cypher version
  - Cypher runtime
  - Cypher connect-components planner
  - Cypher update strategy
  - Cypher expression engine
  - Cypher operator engine
  - Cypher interpreted pipes fallback
  - Cypher replanning
- Profiling a query
- The use of indexes
- Basic query tuning example
- Advanced query tuning example
  - The data set
  - Index-backed property-lookup
  - Index-backed order by
- Planner hints and the USING keyword
  - Introduction
28.1. 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`.

28.1.1. 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:

<table>
<thead>
<tr>
<th>Query option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5</td>
<td>This will force the query to use Neo4j Cypher 3.5.</td>
<td></td>
</tr>
<tr>
<td>4.2</td>
<td>This will force the query to use Neo4j Cypher 4.2.</td>
<td></td>
</tr>
<tr>
<td>4.3</td>
<td>This will force the query to use Neo4j Cypher 4.3.</td>
<td></td>
</tr>
<tr>
<td>4.4</td>
<td>This will force the query to use Neo4j Cypher 4.4. As this is the default version, it is not necessary to use this option explicitly.</td>
<td>✔</td>
</tr>
</tbody>
</table>

In Neo4j 4.4, the support for Cypher 3.5 is provided only at the parser level. The consequence is that some underlying features available in Neo4j 3.5 are no longer available and will result in runtime errors.

Please refer to the discussion in Cypher Compatibility for more information on which features are affected.

28.1.2. Cypher runtime

Using the execution plan, the query is executed — and records returned — by the Cypher 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 the 'faster interpreted' runtime.

**Pipelined**

The pipelined runtime was introduced in Neo4j 4.0 as a replacement for the older compiled runtime used in the Neo4j 3.x versions. It combines some of the advantages of the compiled runtime in a new architecture that allows for support of a wider range of queries.

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).

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>runtime=interpreted</td>
<td>This will force the query planner to use the interpreted runtime.</td>
<td>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.</td>
</tr>
<tr>
<td>runtime=slotted</td>
<td>This will cause the query planner to use the slotted runtime.</td>
<td>This is the default option for all queries which are not supported by runtime=pipelined in Enterprise Edition.</td>
</tr>
</tbody>
</table>
### 28.1.3. Cypher planner

The Cypher planner takes a Cypher query and computes an execution plan that solves it. For any given query there is likely a number of execution plan candidates that each solve the query in a different way. The planner uses a search algorithm to find the execution plan with the lowest estimated execution cost.

This table describes the available planner options:

<table>
<thead>
<tr>
<th>Query option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>planner=cost</td>
<td>Use cost based planning with default limits on plan search space and time.</td>
<td>✔️</td>
</tr>
<tr>
<td>planner=idp</td>
<td>Synonym for planner=cost.</td>
<td></td>
</tr>
<tr>
<td>planner=dp</td>
<td>Use cost based planning without limits on plan search space and time to perform an exhaustive search for the best execution plan. Using this option can significantly increase the planning time of the query.</td>
<td></td>
</tr>
</tbody>
</table>

### 28.1.4. Cypher connect-components planner

One part of the Cypher planner is responsible for combining sub-plans for separate patterns into larger plans - a task referred to as connecting components.

This table describes the available query options for the connect-components planner:
<table>
<thead>
<tr>
<th>Query option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>connectComponentsPlanner=greedy</code></td>
<td>Use a greedy approach when combining sub-plans.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Using this option can significantly reduce the planning time of the query.</td>
<td></td>
</tr>
<tr>
<td><code>connectComponentsPlanner=idp</code></td>
<td>Use the cost based IDP search algorithm when combining sub-plans.</td>
<td>✔️</td>
</tr>
<tr>
<td></td>
<td>Using this option can significantly increase the planning time of the query</td>
<td></td>
</tr>
<tr>
<td></td>
<td>but usually finds better plans.</td>
<td></td>
</tr>
</tbody>
</table>

### 28.1.5. Cypher update strategy

This option affects the eagerness of updating queries.

The possible values are:

<table>
<thead>
<tr>
<th>Query option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>updateStrategy=default</code></td>
<td>Update queries are executed eagerly when needed.</td>
<td>✔️</td>
</tr>
<tr>
<td><code>updateStrategy=eager</code></td>
<td>Update queries are always executed eagerly.</td>
<td></td>
</tr>
</tbody>
</table>

### 28.1.6. Cypher expression engine

This option affects how the runtime evaluates expressions.

The possible values are:

<table>
<thead>
<tr>
<th>Query option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>expressionEngine=default</code></td>
<td>Compile expressions and use the compiled expression engine when needed.</td>
<td>✔️</td>
</tr>
<tr>
<td><code>expressionEngine=interpreted</code></td>
<td>Always use the interpreted expression engine.</td>
<td></td>
</tr>
</tbody>
</table>
### 28.1.7. Cypher operator engine

This query option affects whether the pipelined runtime attempts to generate compiled code for groups of operators.

The possible values are:

<table>
<thead>
<tr>
<th>Query option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expressionEngine=compiled</td>
<td>Always compile expressions and use the compiled expression engine.</td>
</tr>
<tr>
<td></td>
<td>Cannot be used together with runtime=interpreted.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>operatorEngine=default</td>
<td>Attempt to generate compiled operators when applicable.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>operatorEngine=interpreted</td>
<td>Never attempt to generate compiled operators.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>operatorEngine=compiled</td>
<td>Always attempt to generate compiled operators.</td>
</tr>
<tr>
<td></td>
<td>Cannot be used together with runtime=interpreted or runtime=slotted.</td>
</tr>
</tbody>
</table>

### 28.1.8. Cypher interpreted pipes fallback

This query option affects how the pipelined runtime behaves for operators it does not directly support.

The available options are:

<table>
<thead>
<tr>
<th>Query option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>interpretedPipesFallback=default</td>
<td>Equivalent to interpretedPipesFallback=whitelisted_plans_only.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>interpretedPipesFallback=disabled</td>
<td>If the plan contains any operators not supported by the pipelined runtime then another runtime is chosen to execute the entire plan.</td>
</tr>
<tr>
<td></td>
<td>Cannot be used together with runtime=interpreted or runtime=slotted.</td>
</tr>
</tbody>
</table>
### Query option

<table>
<thead>
<tr>
<th>Query option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>interpretedPipesFallback=whitelisted_plans_only</td>
<td>Parts of the execution plan can be executed on another runtime. Only certain operators are allowed to execute on another runtime. Cannot be used together with runtime=interpreted or runtime=slotted.</td>
<td></td>
</tr>
<tr>
<td>interpretedPipesFallback=all</td>
<td>Parts of the execution plan may be executed on another runtime. Any operator is allowed to execute on another runtime. Queries with this option set might produce incorrect results, or fail. Cannot be used together with runtime=interpreted or runtime=slotted.</td>
<td></td>
</tr>
</tbody>
</table>

This setting is experimental, and using it in a production environment is discouraged.

### 28.1.9. Cypher replanning

Cypher replanning occurs in the following circumstances:

- When the query is not in the cache. This can either be when the server is first started or restarted, if the cache has recently been cleared, or if `dbms.query_cache_size` was exceeded.
- When the time has past the `cypher.min_replan_interval` value, and the database statistics have changed more than the `cypher.statistics_divergence_threshold` value.

There may be situations where Cypher query planning can occur at a non-ideal time. For example, when a query must be as fast as possible and a valid plan is already in place.

Replanning is not performed for all queries at once; it is performed in the same thread as running the query, and can block the query. However, replanning one query does not replan any other queries.

There are three different replan options available:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>replan=default</td>
<td>This is the planning and replanning option as described above.</td>
<td>✔</td>
</tr>
</tbody>
</table>
## 28.2. 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. 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.
28.3. The use of indexes

This section describes the query plans when indexes are used in various scenarios.

The task of tuning calls for different indexes depending on what the queries look like. Therefore, it is important to have a fundamental understanding of how the indexes operate. This section describes the query plans that result from different index scenarios.

Node indexes and relationship indexes operate in the same way. Therefore, node and relationship indexes are used interchangeably in this section.

Please refer to Indexes for search performance for instructions on how to create and maintain the indexes themselves.

28.3.1. Index types and predicate compatibility

There are different types of indexes available in Neo4j but they are not all compatible with the same property predicates.

Indexes are commonly used for MATCH and OPTIONAL MATCH clauses that combine a label predicate with a property predicate. Therefore, it is important to know what kind of predicates that can be solved by the different indexes.

**BTREE indexes**

BTREE indexes support all types of predicates:

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>equality check</td>
<td>n.prop = value</td>
</tr>
<tr>
<td>list membership check</td>
<td>n.prop IN list</td>
</tr>
<tr>
<td>existence check</td>
<td>n.prop IS NOT NULL</td>
</tr>
<tr>
<td>range search</td>
<td>n.prop &gt; value</td>
</tr>
<tr>
<td>prefix search</td>
<td>STARTS WITH</td>
</tr>
<tr>
<td>suffix search</td>
<td>ENDS WITH</td>
</tr>
<tr>
<td>substring search</td>
<td>CONTAINS</td>
</tr>
</tbody>
</table>

**TEXT indexes**

TEXT indexes only work for predicates operating on strings. That means that TEXT indexes are only used when it is known that the predicate evaluates to null for all non-string values.

Predicates that only operate on strings are always solvable by a TEXT index:

- STARTS WITH
• ENDS WITH
• CONTAINS

However, other predicates are only used when it is known that the property is compared to a string:

• n.prop = "string"
• n.prop IN ["a", "b", "c"]
• n.prop > "string"

This means that a TEXT index is not able to solve e.g. a.prop = b.prop.

In summary, TEXT indexes support the following predicates:

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>equality check</td>
<td>n.prop = &quot;string&quot;</td>
</tr>
<tr>
<td>list membership check</td>
<td>n.prop IN [&quot;a&quot;, &quot;b&quot;, &quot;c&quot;]</td>
</tr>
<tr>
<td>range search</td>
<td>n.prop &gt; &quot;string&quot;</td>
</tr>
<tr>
<td>prefix search</td>
<td>STARTS WITH</td>
</tr>
<tr>
<td>suffix search</td>
<td>ENDS WITH</td>
</tr>
<tr>
<td>substring search</td>
<td>CONTAINS</td>
</tr>
</tbody>
</table>

28.3.2. Index preference

When multiple indexes are available and able to solve a predicate, there is an order defined that decides which index to use. It is defined as such:

• TEXT indexes are preferred over BTREE indexes for CONTAINS and ENDS WITH.
• BTREE indexes are preferred over TEXT indexes in all other cases.

28.3.3. Node BTREE index example

In the example below, a Person(firstname) node BTREE index is available.

Query

```
MATCH (person:Person {firstname: 'Andy'}) RETURN person
```
28.3.4. Node TEXT index example

In the example below, a Person(surname) node TEXT index is available.

Query

```
MATCH (person:Person {surname: 'Smith'}) RETURN person
```

28.3.5. Relationship BTREE index example

In this example, a KNOWS(since) relationship BTREE index is available.
8.3.6. Relationship TEXT index example

In this example, a `KNOWS(lastMetLocation)` relationship TEXT index is available.

Query

MATCH (person)-[relationship:KNOWS (metIn: 'Malmo')]->(friend) RETURN person, friend
### Query Plan

**Compiler CYPHER 4.4**

**Planner COST**

**Runtime PIPELINED**

**Runtime version 4.4**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimated Rows</td>
<td>Rows</td>
</tr>
<tr>
<td>+ProduceResults</td>
<td>person, friend</td>
</tr>
<tr>
<td>+DirectedRelationshipIndexSeek</td>
<td>TEXT INDEX (person)-[relationship:KNOWS(metIn)]-&gt;(friend) WHERE metIn = $autostring_0</td>
</tr>
</tbody>
</table>

Total database accesses: 3, total allocated memory: 176

### 28.3.7. Multiple available index types

In the example below, both a `Person(middlename)` node TEXT index and a `Person(middlename)` node BTREE index are available. The TEXT node index is chosen.

**Query**

```
MATCH (person:Person {middlename: 'Ron'}) RETURN person
```

562
28.3.8. Equality check using **WHERE** (single-property index)

A query containing equality comparisons of a single indexed property in the **WHERE** clause is backed automatically by the index. It is also possible for a query with multiple **OR** predicates to use multiple indexes, if indexes exist on the properties. For example, if indexes exist on both :Label(p1) and :Label(p2), MATCH (n:Label) WHERE n.p1 = 1 OR n.p2 = 2 RETURN n will use both indexes.

**Query**

```cypher
MATCH (person:Person) WHERE person.firstname = 'Andy' RETURN person
```
28.3.9. Equality check using **WHERE** (composite index)

A query containing equality comparisons for all the properties of a composite index will automatically be backed by the same index. However, the query does not need to have equality on all properties. It can have ranges and existence predicates as well. But in these cases rewrites might happen depending on which properties have which predicates, see composite index limitations. The following query will use the composite index defined earlier:

**Query**

```cypher
MATCH (n:Person) WHERE n.age = 35 AND n.country = 'UK' RETURN n
```

However, the query `MATCH (n:Person) WHERE n.age = 35 RETURN n` will not be backed by the composite index, as the query does not contain a predicate on the `country` property. It will only be backed by an index on the `Person` label and `age` property defined thus: `:Person(age)`; i.e. a single-property index.
28.3.10. Range comparisons using **WHERE** (single-property index)

Single-property indexes are also automatically used for inequality (range) comparisons of an indexed property in the **WHERE** clause.

**Query**

```
MATCH (friend)<-[r:KNOWS]-(person) WHERE r.since < 2011 RETURN friend, person
```

**Query Plan**

```
Compiler CYPHER 4.4
Planner COST
Runtime PIPELINED
Runtime version 4.4

+---------------------------------------+-----------------------------------------------------+----------------+------+
| Operator                              | Details                                             | Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Other |
+---------------------------------------+-----------------------------------------------------+----------------+------+
| +ProduceResults                       | friend, person                                      | 1              | 1    | 0       |                |                        | 0.667      |       |
|                                       |                                                     |                |      |         |              |                         | Fused in Pipeline 0 |
+---------------------------------------+-----------------------------------------------------+----------------+------+
| +DirectedRelationshipIndexSeekByRange | BTREE INDEX (person)-[r:KNOWS(since)]->(friend) WHERE since < $autoint_0 | 1              | 1    | 3       | 112            | 2/1                      | 0.667      |       |
|                                       |                                                     |                |      |         |              |                         | Fused in Pipeline 0 |
```

Total database accesses: 3, total allocated memory: 176

28.3.11. Range comparisons using **WHERE** (composite index)

Composite indexes are also automatically used for inequality (range) comparisons of indexed properties in the **WHERE** clause. Equality or list membership check predicates may precede the range predicate. However, predicates after the range predicate may be rewritten as an existence check predicate and a filter as described in **composite index limitations**.

**Query**

```
MATCH ()-[r:KNOWS]-() WHERE r.since < 2011 AND r.lastMet > 2019 RETURN r.since
```

------

565
### Query Plan

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Memory (Bytes)</th>
<th>Page Cache Hits/Misses</th>
<th>Time (ms)</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td><code>r.since</code></td>
<td>2</td>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>Fused in Pipeline 0</td>
</tr>
<tr>
<td>+Projection</td>
<td>cache[r.since] AS <code>r.since</code></td>
<td>2</td>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>Fused in Pipeline 0</td>
</tr>
<tr>
<td>+Filter</td>
<td>cache[r.lastMet] &gt; $autoint_1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>Fused in Pipeline 0</td>
</tr>
<tr>
<td>+UndirectedRelationshipIndexSeek</td>
<td>BTREE INDEX (anon_0)-[r:KNOWS(since, lastMet)]-(anon_1) WHERE since &lt; $autoint_0 AND lastMet IS NOT NULL, cache[r.since], cache[r.lastMet]</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>112</td>
<td>1/1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total database accesses: 3, total allocated memory: 176

#### 28.3.12. Multiple range comparisons using **WHERE** (single-property index)

When the **WHERE** clause contains multiple inequality (range) comparisons for the same property, these can be combined in a single index range seek.

**Query**

```cypher
MATCH (person:Person) WHERE 10000 < person.highScore < 20000 RETURN person
```
28.3.13. Multiple range comparisons using WHERE (composite index)

When the WHERE clause contains multiple inequality (range) comparisons for the same property, these can be combined in a single index range seek. That single range seek created in the following query will then use the composite index Person(highScore, name) if it exists.

Query

MATCH (person:Person) WHERE 10000 < person.highScore < 20000 AND person.name IS NOT NULL RETURN person
### Query Plan

```
<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>person</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>+NodeIndexSeek</td>
<td>BTREE INDEX person:Person(highScore, name) WHERE highScore &gt; $autoint_0 AND highScore &lt; $autoint_1 A</td>
</tr>
<tr>
<td></td>
<td>ND name IS NOT NULL</td>
</tr>
<tr>
<td></td>
<td>Fused in Pipeline 0</td>
</tr>
</tbody>
</table>
```

Total database accesses: 2, total allocated memory: 176

### 28.3.14. List membership check using **IN** (single-property index)

The **IN** predicate on `r.since` in the following query will use the single-property index `KNOWS(lastMetIn)` if it exists.

#### Query

```
MATCH (person)-[r:KNOWS]->(friend) WHERE r.lastMetIn IN ['Malmo', 'Stockholm'] RETURN person, friend
```
Query Plan

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>person, friend</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

+--------------------------------
| +DirectedRelationshipIndexSeek | BTREE INDEX (person)-[r:KNOWS(lastMetIn)]->(friend) WHERE lastMetIn IN $autolist_0 | 1 | 1 | 4 | 112 | 3/1 | 1.309 |
Fused in Pipeline 0 |

Total database accesses: 4, total allocated memory: 176

28.3.15. List membership check using IN (composite index)

The IN predicates on r.since and r.lastMet in the following query will use the composite index KNOWS(since, lastMet) if it exists.

Query

```
```
### Query Plan

Compiler CYPHER 4.4  
Planner COST  
Runtime PIPELINED  

Runtime version 4.4

| Operator | Details |
|----------------+----------------+----------------+----------------+------------------------+-----------|
| Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Other |
| +ProduceResults | person, friend | 1 | 1 | 0 | | Fused in Pipeline 0 |
| +DirectedRelationshipIndexSeek | BTREE INDEX (person)-[r:KNOWS(since, lastMet)]->(friend) WHERE since IN $ | 1 | 1 | 6 | 112 | 9.974 | Fused in Pipeline 0 |

Total database accesses: 6, total allocated memory: 176

### 28.3.16. Prefix search using STARTS WITH (single-property index)

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

**Query**

```
MATCH (person:Person) WHERE person.firstname STARTS WITH 'And' RETURN person
```
### 28.3.17. Prefix search using STARTS WITH (composite index)

The `STARTS WITH` predicate on `person.firstname` in the following query will use the `Person(firstname,surname)` index, if it exists. Any (non-existence check) predicate on `person.surname` will be rewritten as existence check with a filter. However, if the predicate on `person.firstname` is an equality check then a `STARTS WITH` on `person.surname` would also use the index (without rewrites). More information about how the rewriting works can be found in composite index limitations.

**Query**

```cypher
MATCH (person:Person) WHERE person.firstname STARTS WITH 'And' AND person.surname IS NOT NULL RETURN person
```
28.3.18. Suffix search using ENDS WITH (single-property index)

The ENDS WITH predicate on r.metIn in the following query uses the KNOWS(metIn) index, if it exists. All values stored in the KNOWS(metIn) index are searched, and entries ending with 'mo' are returned. This means that although the search is not optimized to the extent of queries using =, IN, >, < or STARTS WITH, it is still faster than not using an index in the first place.

Query

MATCH (person)-[r:KNOWS]->(friend) WHERE r.metIn ENDS WITH 'mo' RETURN person, friend
28.3.19. Suffix search using ENDS WITH (composite index)

The ENDS WITH predicate on \( r.metIn \) in the following query uses the \( \text{KNOWS}(metIn, lastMetIn) \) index, if it exists. However, it is rewritten as existence check and a filter due to the index not supporting actual suffix searches for composite indexes, this is still faster than not using an index in the first place. Any (non-existence check) predicate on \( \text{KNOWS}.lastMetIn \) is also rewritten as existence check with a filter. More information about how the rewriting works can be found in composite index limitations.

**Query**

```sql
MATCH (person)-[r:KNOWS]->(friend) WHERE r.metIn ENDS WITH 'mo' AND r.lastMetIn IS NOT NULL RETURN person, friend
```
28.3.20. Substring search using CONTAINS (single-property index)

The CONTAINS predicate on person.firstname in the following query will use the Person(firstname) index, if it exists. All values stored in the Person(firstname) index will be searched, and entries containing 'h' will be returned. This means that although the search will not be optimized to the extent of queries using =, IN, >, < or STARTS WITH, it is still faster than not using an index in the first place. Composite indexes are currently not able to support CONTAINS.

Query

MATCH (person:Person) WHERE person.firstname CONTAINS 'h' RETURN person
### Query Plan

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimated Rows</td>
<td>Rows</td>
</tr>
<tr>
<td>+ProduceResults</td>
<td>person</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>+NodeIndexContainsScan</td>
<td>BTREE INDEX person:Person(firstname) WHERE firstname CONTAINS $autostring_0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Total database accesses: 2, total allocated memory: 176

### 28.3.21. Substring search using **CONTAINS** (composite index)

The **CONTAINS** predicate on `person.country` in the following query will use the `Person(country,age)` index, if it exists. However, it will be rewritten as existence check and a filter due to the index not supporting actual suffix searches for composite indexes, this is still faster than not using an index in the first place.

Any (non-existence check) predicate on `person.age` will also be rewritten as existence check with a filter.

More information about how the rewriting works can be found in **composite index limitations**.

#### Query

```cypher
MATCH (person:Person) WHERE person.country CONTAINS '300' AND person.age IS NOT NULL RETURN person
```
28.3.22. Existence check using **IS NOT NULL** (single-property index)

The `r.since IS NOT NULL` predicate in the following query uses the `KNOWS(since)` index, if it exists.

**Query**

```
MATCH (person)-[r:KNOWS]->(friend) WHERE r.since IS NOT NULL RETURN person, friend
```
28.3.23. Existence check using IS NOT NULL (composite index)

The p.firstname IS NOT NULL and p.surname IS NOT NULL predicates in the following query will use the Person(firstname,surname) index, if it exists. Any (non-existence check) predicate on person.surname will be rewritten as existence check with a filter.

Query

MATCH (p:Person) WHERE p.firstname IS NOT NULL AND p.surname IS NOT NULL RETURN p
### Query Plan

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimated Rows</td>
<td>Rows</td>
</tr>
<tr>
<td>ProducesResults</td>
<td>p</td>
</tr>
</tbody>
</table>

Total database accesses: 3, total allocated memory: 176

#### 28.3.24. Spatial distance searches (single-property index)

If a property with point values is indexed, the index is used for spatial distance searches as well as for range queries.

**Query**

```cypher
MATCH ()-[r:KNOWS]->() WHERE point.distance(r.lastMetPoint, point({x: 1, y: 2})) < 2 RETURN r.lastMetPoint
```
### 28.3.25. Spatial distance searches (composite index)

If a property with point values is indexed, the index is used for spatial distance searches as well as for range queries. Any following (non-existence check) predicates (here on property `p.name` for index `:Person(place,name)`) will be rewritten as existence check with a filter.

#### Query

```cypher
MATCH (p:Person) WHERE point.distance(p.place, point({x: 1, y: 2})) < 2 AND p.name IS NOT NULL RETURN p.place
```
28.3.26. Spatial bounding box searches (single-property index)

The ability to do index seeks on bounded ranges works even with the 2D and 3D spatial Point types.

Query

```cypher
MATCH (person:Person) WHERE point.withinBBox(person.location, point({x: 1.2, y: 5.4}), point({x: 1.3, y: 5.5})) RETURN person.firstname
```
### Query Plan

**Compiler** CYPHER 4.4  
**Planner** COST  
**Runtime** PIPELINED

Runtime version 4.4

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimated Rows</td>
<td>Rows</td>
</tr>
<tr>
<td>+ProduceResults</td>
<td><code>person.firstname</code></td>
</tr>
<tr>
<td>+Projection</td>
<td>person.firstname AS <code>person.firstname</code></td>
</tr>
<tr>
<td>+NodeIndexSeekByRange</td>
<td>BTREE INDEX person:Person(location) WHERE point.withinBBox(location, point($autodouble_0, $autodouble_1), point($autodouble_2, $autodouble_3))</td>
</tr>
</tbody>
</table>

Total database accesses: 4, total allocated memory: 176

---

**28.3.27. Spatial bounding box searches (composite index)**

The ability to do index seeks on bounded ranges works even with the 2D and 3D spatial Point types. Any following (non-existence check) predicates (here on property p.firstname for index :Person(place, firstname)) will be rewritten as existence check with a filter. For index :Person(firstname, place), if the predicate on firstname is equality or list membership then the bounded range is handled as a range itself. If the predicate on firstname is anything else then the bounded range is rewritten to existence and filter.

**Query**

```cypher
MATCH (person:Person) WHERE point.withinBBox(person.place, point({x: 1.2, y: 5.4}), point({x: 1.3, y: 5.5})) AND person.firstname IS NOT NULL RETURN person
```
# Query Plan

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimated Rows</td>
<td>Rows</td>
</tr>
<tr>
<td>+ProduceResults</td>
<td>person</td>
</tr>
<tr>
<td>+NodeIndexSeek</td>
<td>BTREE INDEX person:Person(place, firstname) WHERE point.withinBBox(place, point($autodouble_0, $autodouble_1), point($autodouble_2, $autodouble_3)) AND firstname IS NOT NULL</td>
</tr>
</tbody>
</table>
The movies.csv file contains two columns title, released and tagline.

movies.csv

<table>
<thead>
<tr>
<th>title</th>
<th>released</th>
<th>tagline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Something's Gotta Give</td>
<td>1975</td>
<td>null</td>
</tr>
<tr>
<td>Johnny Mnemonic</td>
<td>1995</td>
<td>The hottest data on earth. In the coolest head in town</td>
</tr>
<tr>
<td>The Replacements</td>
<td>2000</td>
<td>&quot;Pain heals, Chicks dig scars... Glory lasts forever&quot;</td>
</tr>
<tr>
<td>The Devil's Advocate</td>
<td>1997</td>
<td>Evil has its winning ways</td>
</tr>
<tr>
<td>The Matrix Revolution</td>
<td>2003,</td>
<td>Everything that has a beginning has an end</td>
</tr>
<tr>
<td>The Matrix Reloaded</td>
<td>2003</td>
<td>Free your mind</td>
</tr>
<tr>
<td>The Matrix, 1999</td>
<td>Welcome to the Real World</td>
<td></td>
</tr>
<tr>
<td>The Matrix Revolution</td>
<td>2003,</td>
<td>Everything that has a beginning has an end</td>
</tr>
<tr>
<td>The Matrix Reloaded</td>
<td>2003</td>
<td>Free your mind</td>
</tr>
<tr>
<td>The Matrix, 1999</td>
<td>Welcome to the Real World</td>
<td></td>
</tr>
<tr>
<td>The Matrix Revolution</td>
<td>2003,</td>
<td>Everything that has a beginning has an end</td>
</tr>
<tr>
<td>The Matrix Reloaded</td>
<td>2003</td>
<td>Free your mind</td>
</tr>
<tr>
<td>The Matrix, 1999</td>
<td>Welcome to the Real World</td>
<td></td>
</tr>
<tr>
<td>Speed Racer, 2008</td>
<td>Speed has no limits</td>
<td></td>
</tr>
<tr>
<td>Cloud Atlas, 2012</td>
<td>Everything is connected</td>
<td></td>
</tr>
<tr>
<td>The Matrix Revolution</td>
<td>2003,</td>
<td>Everything that has a beginning has an end</td>
</tr>
<tr>
<td>The Matrix Reloaded</td>
<td>2003</td>
<td>Free your mind</td>
</tr>
<tr>
<td>The Matrix, 1999</td>
<td>Welcome to the Real World</td>
<td></td>
</tr>
<tr>
<td>Ninja Assassin, 2009</td>
<td>Prepare to enter a secret world of assassins</td>
<td></td>
</tr>
<tr>
<td>V for Vendetta, 2006</td>
<td>Freedom! Forever!</td>
<td></td>
</tr>
<tr>
<td>Speed Racer, 2008</td>
<td>Speed has no limits</td>
<td></td>
</tr>
<tr>
<td>V for Vendetta, 2006</td>
<td>Freedom! Forever!</td>
<td></td>
</tr>
<tr>
<td>Speed Racer, 2008</td>
<td>Speed has no limits</td>
<td></td>
</tr>
<tr>
<td>Cloud Atlas, 2012</td>
<td>Everything is connected</td>
<td></td>
</tr>
<tr>
<td>The Matrix Revolution</td>
<td>2003,</td>
<td>Everything that has a beginning has an end</td>
</tr>
<tr>
<td>The Matrix Reloaded</td>
<td>2003</td>
<td>Free your mind</td>
</tr>
<tr>
<td>The Matrix, 1999</td>
<td>Welcome to the Real World</td>
<td></td>
</tr>
<tr>
<td>Ninja Assassin, 2009</td>
<td>Prepare to enter a secret world of assassins</td>
<td></td>
</tr>
<tr>
<td>V for Vendetta, 2006</td>
<td>Freedom! Forever!</td>
<td></td>
</tr>
<tr>
<td>Speed Racer, 2008</td>
<td>Speed has no limits</td>
<td></td>
</tr>
<tr>
<td>V for Vendetta, 2006</td>
<td>Freedom! Forever!</td>
<td></td>
</tr>
<tr>
<td>Ninja Assassin, 2009</td>
<td>Prepare to enter a secret world of assassins</td>
<td></td>
</tr>
<tr>
<td>Speed Racer, 2008</td>
<td>Speed has no limits</td>
<td></td>
</tr>
<tr>
<td>V for Vendetta, 2006</td>
<td>Freedom! Forever!</td>
<td></td>
</tr>
<tr>
<td>The Matrix Revolution</td>
<td>2003,</td>
<td>Everything that has a beginning has an end</td>
</tr>
<tr>
<td>The Matrix Reloaded</td>
<td>2003</td>
<td>Free your mind</td>
</tr>
<tr>
<td>The Matrix, 1999</td>
<td>Welcome to the Real World</td>
<td></td>
</tr>
<tr>
<td>That Thing You Do, 1996</td>
<td>In every life there comes a time when that thing you dream becomes that thing you do</td>
<td></td>
</tr>
<tr>
<td>The Devil's Advocate</td>
<td>1997</td>
<td>Evil has its winning ways</td>
</tr>
<tr>
<td>The Devil's Advocate</td>
<td>1997</td>
<td>Evil has its winning ways</td>
</tr>
<tr>
<td>Jerry Maguire, 2000</td>
<td>The rest of his life begins now.</td>
<td></td>
</tr>
<tr>
<td>Top Gun, 1986</td>
<td>&quot;I feel the need, the need for speed.&quot;</td>
<td></td>
</tr>
<tr>
<td>A Few Good Men, 1992</td>
<td>&quot;In the heart of the nation's capital, in a courthouse of the U.S. government, one man will stop at nothing to keep his honor, and one will stop at nothing to find the truth.&quot;</td>
<td></td>
</tr>
<tr>
<td>Something's Gotta Give, 1975, null</td>
<td></td>
<td></td>
</tr>
<tr>
<td>One Flew Over the Cuckoo's Nest, 1975</td>
<td>&quot;If he's crazy, what does that make you?&quot;</td>
<td></td>
</tr>
<tr>
<td>Hoffa, 1992</td>
<td>He didn't want law. He wanted justice.</td>
<td></td>
</tr>
<tr>
<td>As Good as It Gets, 1997</td>
<td>A comedy from the heart that goes for the throat.</td>
<td></td>
</tr>
<tr>
<td>A Few Good Men, 1992</td>
<td>&quot;In the heart of the nation's capital, in a courthouse of the U.S. government, one man will stop at nothing to keep his honor, and one will stop at nothing to find the truth.&quot;</td>
<td></td>
</tr>
<tr>
<td>A Few Good Men, 1992</td>
<td>&quot;In the heart of the nation's capital, in a courthouse of the U.S. government, one man will stop at nothing to keep his honor, and one will stop at nothing to find the truth.&quot;</td>
<td></td>
</tr>
<tr>
<td>Apollo 13, 1995</td>
<td>&quot;Houston, we have a problem.&quot;</td>
<td></td>
</tr>
<tr>
<td>Frost/Nixon, 2008</td>
<td>400 million people were waiting for the truth.</td>
<td></td>
</tr>
<tr>
<td>A Few Good Men, 1992</td>
<td>&quot;In the heart of the nation's capital, in a courthouse of the U.S. government, one man will stop at nothing to keep his honor, and one will stop at nothing to find the truth.&quot;</td>
<td></td>
</tr>
</tbody>
</table>
What Dreams May Come, 1998. After life there is more. The end is just the beginning.


As Good as It Gets, 1997. A comedy from the heart that goes for the throat.

You've Got Mail, 1998. At odds in life... in love on-line.

Stand By Me, 1995. "For some, it's the last real taste of innocence, and the first real taste of life. But for everyone, it's the time that memories are made of."

As Good as It Gets, 1997. A comedy from the heart that goes for the throat.


Sleepless in Seattle, 1993. "What if someone you never met, someone you never saw, someone you never knew was the only someone for you?"

You've Got Mail, 1998. At odds in life... in love on-line.

Top Gun, 1986. "I feel the need, the need for speed."

A Few Good Men, 1992. "In the heart of the nation's capital, in a courthouse of the U.S. government, one man will stop at nothing to keep his honor, and one will stop at nothing to find the truth."

A Few Good Men, 1992. "In the heart of the nation's capital, in a courthouse of the U.S. government, one man will stop at nothing to keep his honor, and one will stop at nothing to find the truth."

A Few Good Men, 1992. "In the heart of the nation's capital, in a courthouse of the U.S. government, one man will stop at nothing to keep his honor, and one will stop at nothing to find the truth."


A Few Good Men, 1992. "In the heart of the nation's capital, in a courthouse of the U.S. government, one man will stop at nothing to keep his honor, and one will stop at nothing to find the truth."

A Few Good Men, 1992. "In the heart of the nation's capital, in a courthouse of the U.S. government, one man will stop at nothing to keep his honor, and one will stop at nothing to find the truth."


Ninja Assassin, 2009. Prepare to enter a secret world of assassins

When Harry Met Sally, 1998. At odds in life... in love on-line.

Stand By Me, 1995. "For some, it's the last real taste of innocence, and the first real taste of life. But for everyone, it's the time that memories are made of."

Top Gun, 1986. "I feel the need, the need for speed."

A Few Good Men, 1992. "In the heart of the nation's capital, in a courthouse of the U.S. government, one man will stop at nothing to keep his honor, and one will stop at nothing to find the truth."


Sleepless in Seattle, 1993. "What if someone you never met, someone you never saw, someone you never knew was the only someone for you?"

You've Got Mail, 1998. At odds in life... in love on-line.

Top Gun, 1986. "I feel the need, the need for speed."

A Few Good Men, 1992. "In the heart of the nation's capital, in a courthouse of the U.S. government, one man will stop at nothing to keep his honor, and one will stop at nothing to find the truth."


Stand By Me, 1995. "For some, it's the last real taste of innocence, and the first real taste of life. But for everyone, it's the time that memories are made of."


Stand By Me, 1995. "For some, it's the last real taste of innocence, and the first real taste of life. But for everyone, it's the time that memories are made of."

Stand By Me, 1995. "For some, it's the last real taste of innocence, and the first real taste of life. But for everyone, it's the time that memories are made of."

RescueDawn, 2006. Based on the extraordinary true story of one man's fight for freedom

Stand By Me, 1995. "For some, it's the last real taste of innocence, and the first real taste of life. But for everyone, it's the time that memories are made of."

Cast Away, 2000. "At the edge of the world, his journey begins."


As Good as It Gets, 1997. A comedy from the heart that goes for the throat.

You've Got Mail, 1998. At odds in life... in love on-line.

As Good as It Gets, 1997. A comedy from the heart that goes for the throat.

As Good as It Gets, 1997. A comedy from the heart that goes for the throat.


What Dreams May Come, 1998. After life there is more. The end is just the beginning.
What Dreams May Come, 1998, After life there is more. The end is just the beginning.
Rescue Dawn, 2006, Based on the extraordinary true story of one man's fight for freedom
Bicentennial Man, 1999, One robot's 200 year journey to become an ordinary man.
The Birdcage, 1996, Come as you are
What Dreams May Come, 1998, After life there is more. The end is just the beginning.
Snow Falling on Cedars, 1999, First loves last. Forever.
Ninja Assassin, 2009, Prepare to enter a secret world of assassins
Snow Falling on Cedars, 1999, First loves last. Forever.
The Green Mile, 1999, Walk a mile you'll never forget.
Snow Falling on Cedars, 1999, First loves last. Forever.
You've Got Mail, 1998, At odds in life... in love on-line.
You've Got Mail, 1998, At odds in life... in love on-line.
Rescue Dawn, 2006, Based on the extraordinary true story of one man's fight for freedom
You've Got Mail, 1998, At odds in life... in love on-line.
A League of Their Own, 1992, Once in a lifetime you get a chance to do something different.
The Polar Express, 2004, This Holiday Season... Believe
Cast Away, 2000, "At the edge of the world, his journey begins."
Apollo 13, 1995, "Houston, we have a problem."
The Green Mile, 1999, Walk a mile you'll never forget.
The Da Vinci Code, 2006, Break The Codes
Cloud Atlas, 2012, Everything is connected
That Thing You Do, 1996, In every life there comes a time when that thing you dream becomes that thing you do
Joe Versus the Volcano, 1990, "A story of love, lava and burning desire."
Sleepless in Seattle, 1993, "What if someone you never met, someone you never saw, someone you never knew was the only someone for you?"
You've Got Mail, 1998, At odds in life... in love on-line.
That Thing You Do, 1996, In every life there comes a time when that thing you dream becomes that thing you do
Sleepless in Seattle, 1993, "What if someone you never met, someone you never saw, someone you never knew was the only someone for you?"
You've Got Mail, 1998, At odds in life... in love on-line.
When Harry Met Sally, 1998, At odds in life... in love on-line.
When Harry Met Sally, 1998, At odds in life... in love on-line.
Sleepless in Seattle, 1993, "What if someone you never met, someone you never saw, someone you never knew was the only someone for you?"
Sleepless in Seattle, 1993, "What if someone you never met, someone you never saw, someone you never knew was the only someone for you?"
Sleepless in Seattle, 1993, "What if someone you never met, someone you never saw, someone you never knew was the only someone for you?"
A League of Their Own, 1992, Once in a lifetime you get a chance to do something different.
Sleepless in Seattle, 1993, "What if someone you never met, someone you never saw, someone you never knew was the only someone for you?"
Joe Versus the Volcano, 1990, "A story of love, lava and burning desire."
The Birdcage, 1996, Come as you are
Joe Versus the Volcano, 1990, "A story of love, lava and burning desire."
When Harry Met Sally, 1998, At odds in life... in love on-line.
When Harry Met Sally, 1998, At odds in life... in love on-line.
That Thing You Do, 1996, In every life there comes a time when that thing you dream becomes that thing you do
The Replacements, 2000, "Pain heals, Chicks dig scars... Glory lasts forever"
Unforgiven, 1992, "It's a hell of a thing, killing a man"
The Birdcage, 1996, Come as you are
The Replacements, 2000, "Pain heals, Chicks dig scars... Glory lasts forever"
The Replacements, 2000, "Pain heals, Chicks dig scars... Glory lasts forever"
Rescue Dawn, 2006, Based on the extraordinary true story of one man's fight for freedom
Twister, 1996, Don't Breathe. Don't Look Back.
Rescue Dawn, 2006, Based on the extraordinary true story of one man's fight for freedom
The Birdcage, 1996, Come as you are
Unforgiven, 1992, "It's a hell of a thing, killing a man"
Unforgiven, 1992, "It's a hell of a thing, killing a man"
Unforgiven, 1992, "It's a hell of a thing, killing a man"
Johnny Mnemonic, 1995, The hottest data on earth. In the coolest head in town
Johnny Mnemonic, 1995, The hottest data on earth. In the coolest head in town
Johnny Mnemonic, 1995, The hottest data on earth. In the coolest head in town
Johnny Mnemonic, 1995, The hottest data on earth. In the coolest head in town
Cloud Atlas, 2012, Everything is connected
Cloud Atlas, 2012, Everything is connected
Cloud Atlas, 2012, Everything is connected
The Da Vinci Code, 2006, Break The Codes
The Da Vinci Code, 2006, Break The Codes
The Da Vinci Code, 2006, Break The Codes
Apollo 13, 1995, "Houston, we have a problem."
Frost/Nixon, 2008, 400 million people were waiting for the truth.
The Da Vinci Code, 2006, Break The Codes
V for Vendetta, 2006, Freedom! Forever!
V for Vendetta, 2006, Freedom! Forever!
V for Vendetta, 2006, Freedom! Forever!
Ninja Assassin, 2009, Prepare to enter a secret world of assassins
Speed Racer, 2008, Speed has no limits
V for Vendetta, 2006, Freedom! Forever!
Speed Racer, 2008, Speed has no limits
Speed Racer, 2008, Speed has no limits
Speed Racer, 2008, Speed has no limits
Speed Racer, 2008, Speed has no limits
Ninja Assassin, 2009, Prepare to enter a secret world of assassins
Speed Racer, 2008, Speed has no limits
Ninja Assassin, 2009, Prepare to enter a secret world of assassins
The Green Mile, 1999, Walk a mile you'll never forget.
The Green Mile, 1999, Walk a mile you'll never forget.
The Green Mile, 1999, Walk a mile you'll never forget.
Frost/Nixon, 2008, 400 million people were waiting for the truth.
The Green Mile, 1999, Walk a mile you'll never forget.
Apollo 13, 1995, "Houston, we have a problem."
The Green Mile, 1999, Walk a mile you'll never forget.
The Green Mile, 1999, Walk a mile you'll never forget.
The Green Mile, 1999, Walk a mile you'll never forget.
Frost/Nixon, 2008, 400 million people were waiting for the truth.
Frost/Nixon, 2008, 400 million people were waiting for the truth.
Bicentennial Man, 1999, One robot's 200 year journey to become an ordinary man.
Frost/Nixon, 2008, 400 million people were waiting for the truth.
One Flew Over the Cuckoo's Nest, 1975, "If he's crazy, what does that make you?"
Hoffa, 1992, He didn't want law. He wanted justice.
Hoffa, 1992, He didn't want law. He wanted justice.
Hoffa, 1992, He didn't want law. He wanted justice.
Apollo 13, 1995, "Houston, we have a problem."
A League of Their Own, 1992, Once in a lifetime you get a chance to do something different.
Twister, 1996, Don't Breathe. Don't Look Back.
Apollo 13, 1995, "Houston, we have a problem."
Twister, 1996, Don't Breathe. Don't Look Back.
Twister, 1996, Don't Breathe. Don't Look Back.
The Polar Express, 2004, This Holiday Season. Believe
Cast Away, 2000, "At the edge of the world, his journey begins."
One Flew Over the Cuckoo's Nest, 1975, "If he's crazy, what does that make you?"
Something's Gotta Give, 1975, null
Something's Gotta Give, 1975, null
Something's Gotta Give, 1975, null
Something's Gotta Give, 1975, null
Bicentennial Man, 1999, One robot's 200 year journey to become an ordinary man.
A League of Their Own, 1992, Once in a lifetime you get a chance to do something different.
A League of Their Own, 1992, Once in a lifetime you get a chance to do something different.
A League of Their Own, 1992, Once in a lifetime you get a chance to do something different.
The Replacements, 2000, "Pain heals, Chicks dig scars... Glory lasts forever"
The Da Vinci Code, 2006, Break The Codes
The Birdcage, 1996, Come as you are
Unforgiven, 1992, "It's a hell of a thing, killing a man"
The Replacements, 2000, "Pain heals, Chicks dig scars... Glory lasts forever"
Cloud Atlas, 2012, Everything is connected
The Da Vinci Code, 2006, Break The Codes
The Replacements, 2000, "Pain heals, Chicks dig scars... Glory lasts forever"

Actors

The actors.csv file contains two columns title, roles, name and born.

The content of the actors.csv file:
Twister, Dr. Jo Harding, Helen Hunt, 1963
As Good as It Gets, Carol Connelly, Helen Hunt, 1963
You've Got Mail, Frank Navasky, Greg Kinnear, 1963
As Good as It Gets, Simon Bishop, Greg Kinnear, 1963
What Dreams May Come, Simon Bishop, Annabella Sciorra, 1960
Snow Falling on Cedars, Nels Gudmundsson, Max von Sydow, 1929
What Dreams May Come, The Tracker, Max von Sydow, 1929
What Dreams May Come, The Face, Werner Herzog, 1942
Bicentennial Man, Andrew Marin, Robin Williams, 1951
The Birdcage, Armand Goldman, Robin Williams, 1951
What Dreams May Come, Chris Nielsen, Robin Williams, 1951
Snow Falling on Cedars, Ishmael Chambers, Ethan Hawke, 1970
Ninja Assassin, Takeshi, Rick Yune, 1971
Snow Falling on Cedars, Kazuo Miyamoto, Rick Yune, 1971
The Green Mile, Warden Hal Moores, James Cromwell, 1940
Snow Falling on Cedars, Judge Fielding, James Cromwell, 1940
You've Got Mail, Patricia Eden, Parker Posey, 1968
You've Got Mail, Kevin Jackson, Dave Chappelle, 1973
Rescue Dawn, Duane, Steve Zahn, 1967
You've Got Mail, George Pappas, Steve Zahn, 1967
A League of Their Own, Jimmy Dugan, Tom Hanks, 1956
The Polar Express, Hero Boy, Father, Conductor, Hobo, Scrooge, Santa Claus, Tom Hanks, 1956
Cast Away, Chuck Noland, Tom Hanks, 1956
Apollo 13, Jim Lovell, Tom Hanks, 1956
The Green Mile, Paul Edgecomb, Tom Hanks, 1956
The Da Vinci Code, Dr. Robert Langdon, Tom Hanks, 1956
Cloud Atlas, Zachry, Dr. Henry Goose, Isaac Sachs, Dermot Hoggins, Tom Hanks, 1956
That Thing You Do, Mr. White, Tom Hanks, 1956
Joe Versus the Volcano, Joe Banks, Tom Hanks, 1956
Sleepless in Seattle, Suzy, Rita Wilson, 1956
Sleepless in Seattle, Walter, Bill Pullman, 1953
Sleepless in Seattle, Greg, Victor Garber, 1949
A League of Their Own, Doris Murphy, Rosie O'Donnell, 1962
Sleepless in Seattle, Becky, Rosie O'Donnell, 1962
The Birdcage, Albert Goldman, Nathan Lane, 1956
Joe Versus the Volcano, Baw, Nathan Lane, 1956
When Harry Met Sally, Harry Burns, Billy Crystal, 1948
When Harry Met Sally, Marie, Carrie Fisher, 1956
When Harry Met Sally, Jess, Bruno Kirby, 1949
That Thing You Do, Faye Dolan, Liv Tyler, 1977
The Replacements, Annabelle Farrell, Brooke Langton, 1970
Unforgiven, Little Bill Daggett, Gene Hackman, 1930
The Birdcage, Sen. Kevin Keeley, Gene Hackman, 1930
The Replacements, Jimmy McGinty, Gene Hackman, 1930
The Replacements, Clifford Franklin, Orlando Jones, 1968
Rescue Dawn, Dieter Dengler, Christian Bale, 1974
Twister, Eddie, Zach Grenier, 1954
Rescue Dawn, Squad Leader, Zach Grenier, 1954
Unforgiven, English Bob, Richard Harris, 1939
Unforgiven, Bill Munny, Clint Eastwood, 1939
Johnny Mnemonic, Takahashi, Takeshi Kitano, 1947
Johnny Mnemonic, Jane, Dina Meyer, 1968
Johnny Mnemonic, J-Bone, Ice-T, 1958
Cloud Atlas, Luisa Rey, Jocasta Ays, Ovid, Meronym, Halle Berry, 1966
Cloud Atlas, Vyvyan Ays, Captain Molyneux, Timothy Cavendish, Jim Broadbent, 1949
The Da Vinci Code, Sir Leight Teabing, Ian McKellen, 1939
The Da Vinci Code, Sophie Neveu, Audrey Tautou, 1976
The Da Vinci Code, Silas, Paul Bettany, 1971
V for Vendetta, Evey Hammond, Natalie Portman, 1981
V for Vendetta, Eric Finch, Stephen Rea, 1946
V for Vendetta, High Chancellor Adam Butler, John Hurt, 1940
Ninja Assassin, Ryan Maslow, Ben Miles, 1967
Speed Racer, Cass Jones, Ben Miles, 1967
V for Vendetta, Dascomb, Ben Miles, 1967
Speed Racer, Speed Racer, Emile Hirsch, 1985
Speed Racer, Pops, John Goodman, 1960
Speed Racer, Mom, Susan Sarandon, 1946
Speed Racer, Racer X, Matthew Fox, 1966
Speed Racer, Trixie, Christina Ricci, 1980
Ninja Assassin, Raizo, Rain, 1982
Speed Racer, Taejo Togokahn, Rain, 1982
Ninja Assassin, Mika Coretti, Naomie Harris, null
The Green Mile, John Coffey, Michael Clarke Duncan, 1957
The Green Mile, Brutus 'Brutal' Howell, David Morse, 1953
Directors

The directors.csv file contains two columns title, name and born.

The content of the directors.csv file:
28.4.2. Prerequisites

The example uses the Linux or macOS tarball installation. It assumes that your current work directory is the `<neo4j-home>` directory of the tarball installation, and the CSV files are placed in the default import directory.

- For the default directory of other installations see, Operations Manual → File locations.
- The import location can be configured with Operations Manual → dbms.directories.import.

28.4.3. Importing the data

Import the movies.csv file
LOAD CSV WITH HEADERS FROM 'file:///movies.csv' AS line
MERGE (m:Movie {title: line.title})
ON CREATE SET
  m.released = toInteger(line.released),
  m.tagline = line.tagline

Added 38 nodes, Set 114 properties, Added 38 labels

Import the `actors.csv` file

LOAD CSV WITH HEADERS FROM 'file:///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)

Added 102 nodes, Created 172 relationships, Set 375 properties, Added 102 labels

Import the `directors.csv` file

LOAD CSV WITH HEADERS FROM 'file:///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)

Added 23 nodes, Created 44 relationships, Set 46 properties, Added 23 labels

28.4.4. Profile query

Let’s say you 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 [Query tuning](#) but in this case we’re going to prefix our query with `PROFILE`:

```
PROFILE MATCH (p {name: 'Tom Hanks'})
RETURN p
```
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 \texttt{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 \texttt{Operator} column we'll see that \texttt{AllNodesScan} has been used which means that the query planner scanned through all the nodes in the database.

The \texttt{Filter} operator which will check the \texttt{name} property on each of the nodes passed through by \texttt{AllNodesScan}.

This seems like an inefficient way of finding 'Tom Hanks' given that we are looking at many nodes that aren't even people and therefore aren't what we're looking for.

The solution to this problem is that whenever we're looking for a node we should specify a label to help the query planner narrow down the search space.

For this query we'd need to add a \texttt{Person} label.

\begin{verbatim}
MATCH (p:Person { name: 'Tom Hanks'})
RETURN p
\end{verbatim}

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:
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 FOR (p:Person)
ON (p.name)
```

Added 1 indexes

```
CALL db.awaitIndexes
```

Now if we run the query again it will run more quickly:
Let's profile the query to see why that is:

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

Our execution plan is down to a single row and uses the Node Index Seek operator which does an index seek (see Indexes for search performance) to find the appropriate node.

### 28.5. Advanced query tuning example

This section describes some more subtle optimizations based on native index capabilities.

One of the most important and useful ways of optimizing Cypher queries involves creating appropriate indexes. This is described in more detail in Indexes for search performance, and demonstrated in Basic query tuning example. In summary, an index will be based on the combination of a Label and a property. Any Cypher query that searches for nodes with a specific label and some predicate on the property (equality, range or existence) will be planned to use the index if the cost planner deems that to be the most efficient solution.

In order to benefit from enhancements provided by native indexes, it is useful to understand when index-backed property lookup and index-backed order by will come into play. Let's explain how to use these features with a more advanced query tuning example.

---

If you are upgrading an existing store to 4.4.10, it may be necessary to drop and re-create existing indexes. For information on native index support and upgrade considerations regarding indexes, see Operations Manual → Indexes.
28.5.1. The data set

In this section, examples demonstrate the impact native indexes can have on query performance under certain conditions. You will use a movies dataset to illustrate this more advanced query tuning.

In this tutorial, you import data from the following CSV files:

- movies.csv
- actors.csv
- directors.csv

Movies

The movies.csv file contains two columns title, released and tagline.

The content of the movies.csv file:

<table>
<thead>
<tr>
<th>title</th>
<th>released</th>
<th>tagline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Something’s Gotta Give</td>
<td>1975</td>
<td>null</td>
</tr>
<tr>
<td>Johnny Mnemonic</td>
<td>1995</td>
<td>The hottest data on earth. In the coolest head in town</td>
</tr>
<tr>
<td>The Replacements</td>
<td>2000</td>
<td>“Pain heals, Chicks dig scars... Glory lasts forever”</td>
</tr>
<tr>
<td>The Devil’s Advocate</td>
<td>1997</td>
<td>Evil has its winning ways</td>
</tr>
<tr>
<td>The Matrix Revolutions</td>
<td>2003</td>
<td>Everything that has a beginning has an end</td>
</tr>
<tr>
<td>The Matrix Reloaded</td>
<td>2003</td>
<td>Free your mind</td>
</tr>
<tr>
<td>The Matrix, 1999</td>
<td>Welcome to the Real World</td>
<td></td>
</tr>
<tr>
<td>The Matrix Revolutions</td>
<td>2003</td>
<td>Everything that has a beginning has an end</td>
</tr>
<tr>
<td>The Matrix Reloaded</td>
<td>2003</td>
<td>Free your mind</td>
</tr>
<tr>
<td>The Matrix, 1999</td>
<td>Welcome to the Real World</td>
<td></td>
</tr>
<tr>
<td>The Matrix Revolutions</td>
<td>2003</td>
<td>Everything that has a beginning has an end</td>
</tr>
<tr>
<td>The Matrix Reloaded</td>
<td>2003</td>
<td>Free your mind</td>
</tr>
<tr>
<td>The Matrix, 1999</td>
<td>Welcome to the Real World</td>
<td></td>
</tr>
<tr>
<td>V for Vendetta</td>
<td>2006</td>
<td>Freedom! Forever!</td>
</tr>
<tr>
<td>Cloud Atlas</td>
<td>2012</td>
<td>Everything is connected</td>
</tr>
<tr>
<td>The Matrix Revolutions</td>
<td>2003</td>
<td>Everything that has a beginning has an end</td>
</tr>
<tr>
<td>The Matrix Reloaded</td>
<td>2003</td>
<td>Free your mind</td>
</tr>
<tr>
<td>The Matrix, 1999</td>
<td>Welcome to the Real World</td>
<td></td>
</tr>
<tr>
<td>Speed Racer</td>
<td>2008</td>
<td>Speed has no limits</td>
</tr>
<tr>
<td>Cloud Atlas</td>
<td>2012</td>
<td>Everything is connected</td>
</tr>
<tr>
<td>The Matrix Revolutions</td>
<td>2003</td>
<td>Everything that has a beginning has an end</td>
</tr>
<tr>
<td>The Matrix Reloaded</td>
<td>2003</td>
<td>Free your mind</td>
</tr>
<tr>
<td>The Matrix, 1999</td>
<td>Welcome to the Real World</td>
<td></td>
</tr>
<tr>
<td>Ninja Assassin</td>
<td>2009</td>
<td>Prepare to enter a secret world of assassins</td>
</tr>
<tr>
<td>V for Vendetta</td>
<td>2006</td>
<td>Freedom! Forever!</td>
</tr>
<tr>
<td>Speed Racer</td>
<td>2008</td>
<td>Speed has no limits</td>
</tr>
<tr>
<td>V for Vendetta</td>
<td>2006</td>
<td>Freedom! Forever!</td>
</tr>
<tr>
<td>Speed Racer</td>
<td>2008</td>
<td>Speed has no limits</td>
</tr>
<tr>
<td>Cloud Atlas</td>
<td>2012</td>
<td>Everything is connected</td>
</tr>
<tr>
<td>The Matrix Revolutions</td>
<td>2003</td>
<td>Everything that has a beginning has an end</td>
</tr>
<tr>
<td>The Matrix Reloaded</td>
<td>2003</td>
<td>Free your mind</td>
</tr>
<tr>
<td>The Matrix, 1999</td>
<td>Welcome to the Real World</td>
<td></td>
</tr>
<tr>
<td>Ninja Assassin</td>
<td>2009</td>
<td>Prepare to enter a secret world of assassins</td>
</tr>
<tr>
<td>V for Vendetta</td>
<td>2006</td>
<td>Freedom! Forever!</td>
</tr>
<tr>
<td>Speed Racer</td>
<td>2008</td>
<td>Speed has no limits</td>
</tr>
<tr>
<td>V for Vendetta</td>
<td>2006</td>
<td>Freedom! Forever!</td>
</tr>
<tr>
<td>Speed Racer</td>
<td>2008</td>
<td>Speed has no limits</td>
</tr>
<tr>
<td>Cloud Atlas</td>
<td>2012</td>
<td>Everything is connected</td>
</tr>
<tr>
<td>The Matrix Revolutions</td>
<td>2003</td>
<td>Everything that has a beginning has an end</td>
</tr>
<tr>
<td>The Matrix Reloaded</td>
<td>2003</td>
<td>Free your mind</td>
</tr>
<tr>
<td>The Matrix, 1999</td>
<td>Welcome to the Real World</td>
<td></td>
</tr>
<tr>
<td>Ninja Assassin</td>
<td>2009</td>
<td>Prepare to enter a secret world of assassins</td>
</tr>
<tr>
<td>V for Vendetta</td>
<td>2006</td>
<td>Freedom! Forever!</td>
</tr>
<tr>
<td>Speed Racer</td>
<td>2008</td>
<td>Speed has no limits</td>
</tr>
<tr>
<td>V for Vendetta</td>
<td>2006</td>
<td>Freedom! Forever!</td>
</tr>
<tr>
<td>The Matrix Revolutions</td>
<td>2003</td>
<td>Everything that has a beginning has an end</td>
</tr>
<tr>
<td>The Matrix Reloaded</td>
<td>2003</td>
<td>Free your mind</td>
</tr>
<tr>
<td>The Matrix, 1999</td>
<td>Welcome to the Real World</td>
<td></td>
</tr>
<tr>
<td>That Thing You Do</td>
<td>1996</td>
<td>In every life there comes a time when that thing you dream becomes that thing you do</td>
</tr>
<tr>
<td>The Devil’s Advocate</td>
<td>1997</td>
<td>Evil has its winning ways</td>
</tr>
</tbody>
</table>
Stand By Me, 1995, "For some, it's the last real taste of innocence, and the first real taste of life. But
for everyone, it's the time that memories are made of."

Jerry Maguire, 2000, The rest of his life begins now.

Top Gun, 1986, "I feel the need, the need for speed."

A Few Good Men, 1992, "In the heart of the nation's capital, in a courthouse of the U.S. government, one man
will stop at nothing to keep his honor, and one will stop at nothing to find the truth."

Apollo 13, 1995, "Houston, we have a problem."

Frost/Nixon, 2008, 400 million people were waiting for the truth.

A Few Good Men, 1992, "In the heart of the nation's capital, in a courthouse of the U.S. government, one man
will stop at nothing to keep his honor, and one will stop at nothing to find the truth."

Top Gun, 1986, "I feel the need, the need for speed."

As Good As It Gets, 1997, A comedy from the heart that goes for the throat.

Hoffa, 1992, He didn't want law. He wanted justice.

A Few Good Men, 1992, "In the heart of the nation's capital, in a courthouse of the U.S. government, one man
will stop at nothing to keep his honor, and one will stop at nothing to find the truth."

Stand By Me, 1995, "For some, it's the last real taste of innocence, and the first real taste of life. But
for everyone, it's the time that memories are made of."

Jerry Maguire, 2000, The rest of his life begins now.

Top Gun, 1986, "I feel the need, the need for speed."

A Few Good Men, 1992, "In the heart of the nation's capital, in a courthouse of the U.S. government, one man
will stop at nothing to keep his honor, and one will stop at nothing to find the truth."

Frost/Nixon, 2008, 400 million people were waiting for the truth.

As Good As It Gets, 1997, A comedy from the heart that goes for the throat.

Hoffa, 1992, He didn't want law. He wanted justice.

A Few Good Men, 1992, "In the heart of the nation's capital, in a courthouse of the U.S. government, one man
will stop at nothing to keep his honor, and one will stop at nothing to find the truth."

Stand By Me, 1995, "For some, it's the last real taste of innocence, and the first real taste of life. But
for everyone, it's the time that memories are made of."

Jerry Maguire, 2000, The rest of his life begins now.

Top Gun, 1986, "I feel the need, the need for speed."

A Few Good Men, 1992, "In the heart of the nation's capital, in a courthouse of the U.S. government, one man
will stop at nothing to keep his honor, and one will stop at nothing to find the truth."

Frost/Nixon, 2008, 400 million people were waiting for the truth.

A Few Good Men, 1992, "In the heart of the nation's capital, in a courthouse of the U.S. government, one man
will stop at nothing to keep his honor, and one will stop at nothing to find the truth."

Hoffa, 1992, He didn't want law. He wanted justice.

A Few Good Men, 1992, "In the heart of the nation's capital, in a courthouse of the U.S. government, one man
will stop at nothing to keep his honor, and one will stop at nothing to find the truth."

Top Gun, 1986, "I feel the need, the need for speed."

As Good As It Gets, 1997, A comedy from the heart that goes for the throat.

Hoffa, 1992, He didn't want law. He wanted justice.

A Few Good Men, 1992, "In the heart of the nation's capital, in a courthouse of the U.S. government, one man
will stop at nothing to keep his honor, and one will stop at nothing to find the truth."

Stand By Me, 1995, "For some, it's the last real taste of innocence, and the first real taste of life. But
for everyone, it's the time that memories are made of."

Jerry Maguire, 2000, The rest of his life begins now.

Top Gun, 1986, "I feel the need, the need for speed."

As Good As It Gets, 1997, A comedy from the heart that goes for the throat.

Jerry Maguire, 2000, The rest of his life begins now.

Sleepless In Seattle, 1993, "What if someone you never met, someone you never saw, someone you never knew
was the only someone for you?"

Top Gun, 1986, "I feel the need, the need for speed."

As Good As It Gets, 1997, A comedy from the heart that goes for the throat.

Top Gun, 1986, "I feel the need, the need for speed."

Jerry Maguire, 2000, The rest of his life begins now.

Sleepless In Seattle, 1993, "What if someone you never met, someone you never saw, someone you never knew
was the only someone for you?"

Top Gun, 1986, "I feel the need, the need for speed."

As Good As It Gets, 1997, A comedy from the heart that goes for the throat.

Top Gun, 1986, "I feel the need, the need for speed."

Jerry Maguire, 2000, The rest of his life begins now.

Sleepless In Seattle, 1993, "What if someone you never met, someone you never saw, someone you never knew
was the only someone for you?"

For Vendetta, 2006, Freedom! Forever!

A Few Good Men, 1992, "In the heart of the nation's capital, in a courthouse of the U.S. government, one man
will stop at nothing to keep his honor, and one will stop at nothing to find the truth."

Jerry Maguire, 2000, The rest of his life begins now.

Sleepless In Seattle, 1993, "What if someone you never met, someone you never saw, someone you never knew
was the only someone for you?"

Runaway Jury, 2003, "No one is above the law."

For Vendetta, 2006, Freedom! Forever!

A Few Good Men, 1992, "In the heart of the nation's capital, in a courthouse of the U.S. government, one man
will stop at nothing to keep his honor, and one will stop at nothing to find the truth."

Hacky Sack, 1998, "You're too young to play hacky sack."

For Vendetta, 2006, Freedom! Forever!

A Few Good Men, 1992, "In the heart of the nation's capital, in a courthouse of the U.S. government, one man
will stop at nothing to keep his honor, and one will stop at nothing to find the truth."

The Devil's Advocate, 1997, Evil has its winning ways

For Vendetta, 2006, Freedom! Forever!

A Few Good Men, 1992, "In the heart of the nation's capital, in a courthouse of the U.S. government, one man
will stop at nothing to keep his honor, and one will stop at nothing to find the truth."

The Devil's Advocate, 1997, Evil has its winning ways

Top Gun, 1986, "I feel the need, the need for speed."

As Good As It Gets, 1997, A comedy from the heart that goes for the throat.

Hoffa, 1992, He didn't want law. He wanted justice.

A Few Good Men, 1992, "In the heart of the nation's capital, in a courthouse of the U.S. government, one man
will stop at nothing to keep his honor, and one will stop at nothing to find the truth."

Eyes Wide Shut, 1999, "I feel the need, the need for speed."

As Good As It Gets, 1997, A comedy from the heart that goes for the throat.

Hoffa, 1992, He didn't want law. He wanted justice.

A Few Good Men, 1992, "In the heart of the nation's capital, in a courthouse of the U.S. government, one man
will stop at nothing to keep his honor, and one will stop at nothing to find the truth."

Eyes Wide Shut, 1999, "I feel the need, the need for speed."

As Good As It Gets, 1997, A comedy from the heart that goes for the throat.

Hoffa, 1992, He didn't want law. He wanted justice.

A Few Good Men, 1992, "In the heart of the nation's capital, in a courthouse of the U.S. government, one man
will stop at nothing to keep his honor, and one will stop at nothing to find the truth."

Eyes Wide Shut, 1999, "I feel the need, the need for speed."

As Good As It Gets, 1997, A comedy from the heart that goes for the throat.

Hoffa, 1992, He didn't want law. He wanted justice.

A Few Good Men, 1992, "In the heart of the nation's capital, in a courthouse of the U.S. government, one man
will stop at nothing to keep his honor, and one will stop at nothing to find the truth."

Eyes Wide Shut, 1999, "I feel the need, the need for speed."

As Good As It Gets, 1997, A comedy from the heart that goes for the throat.
for everyone, it's the time that memories are made of.

Stand By Me,1995,"For some, it's the last real taste of innocence, and the first real taste of life. But for everyone, it's the time that memories are made of."

RescueDawn,2006, Based on the extraordinary true story of one man's fight for freedom

Stand By Me,1995,"For some, it's the last real taste of innocence, and the first real taste of life. But for everyone, it's the time that memories are made of."

Cast Away,2000,"At the edge of the world, his journey begins."

Twister,1996, Don't Breathe. Don't Look Back.

As Good as It Gets,1997, A comedy from the heart that goes for the throat.

You've Got Mail,1998, At odds in life... in love on-line.

As Good as It Gets,1997, A comedy from the heart that goes for the throat.

What Dreams May Come,1998, After life there is more. The end is just the beginning.

Snow Falling on Cedars,1999, First loves last. Forever.

What Dreams May Come,1998, After life there is more. The end is just the beginning.

What Dreams May Come,1998, After life there is more. The end is just the beginning.

RescueDawn,2006, Based on the extraordinary true story of one man's fight for freedom

Bicentennial Man,1999, One robot's 200 year journey to become an ordinary man.

The Birdcage,1996, Come as you are

What Dreams May Come,1998, After life there is more. The end is just the beginning.

What Dreams May Come,1998, After life there is more. The end is just the beginning.

Snow Falling on Cedars,1999, First loves last. Forever.

Ninja Assassin,2009, Prepare to enter a secret world of assassins

Snow Falling on Cedars,1999, First loves last. Forever.

The Green Mile,1999, Walk a mile you'll never forget.

Snow Falling on Cedars,1999, First loves last. Forever.

Snow Falling on Cedars,1999, First loves last. Forever.

You've Got Mail,1998, At odds in life... in love on-line.

You've Got Mail,1998, At odds in life... in love on-line.

RescueDawn,2006, Based on the extraordinary true story of one man's fight for freedom

You've Got Mail,1998, At odds in life... in love on-line.

A League of Their Own,1992, Once in a lifetime you get a chance to do something different.

The Polar Express,2004, This Holiday Season... Believe


Cast Away,2000, "At the edge of the world, his journey begins."

Apollo 13,1995, "Houston, we have a problem."

The Green Mile,1999, Walk a mile you'll never forget.

The Da Vinci Code,2006, Break The Codes

Cloud Atlas,2012, Everything is connected

That Thing You Do,1996, In every life there comes a time when that thing you dream becomes that thing you do

Joe Versus the Volcano,1990, "A story of love, lava and burning desire."

Sleepless in Seattle,1993, "What if someone you never met, someone you never saw, someone you never knew was the only someone for you?"

You've Got Mail,1998, At odds in life... in love on-line.

That Thing You Do,1996, In every life there comes a time when that thing you dream becomes that thing you do

Sleepless in Seattle,1993, "What if someone you never met, someone you never saw, someone you never knew was the only someone for you?"

You've Got Mail,1998, At odds in life... in love on-line.

When Harry Met Sally,1998, At odds in life... in love on-line.

Sleepless in Seattle,1993, "What if someone you never met, someone you never saw, someone you never knew was the only someone for you?"

Sleepless in Seattle,1993, "What if someone you never met, someone you never saw, someone you never knew was the only someone for you?"

Sleepless in Seattle,1993, "What if someone you never met, someone you never saw, someone you never knew was the only someone for you?"

Sleepless in Seattle,1993, "What if someone you never met, someone you never saw, someone you never knew was the only someone for you?"

A League of Their Own,1992, Once in a lifetime you get a chance to do something different.

Sleepless in Seattle,1993, "What if someone you never met, someone you never saw, someone you never knew was the only someone for you?"

Joe Versus the Volcano, 1990, "A story of love, lava and burning desire."

The Birdcage,1996, Come as you are

Joe Versus the Volcano,1990, "A story of love, lava and burning desire."

When Harry Met Sally,1998, At odds in life... in love on-line.

When Harry Met Sally,1998, At odds in life... in love on-line.

When Harry Met Sally,1998, At odds in life... in love on-line.

That Thing You Do,1996, In every life there comes a time when that thing you dream becomes that thing you do

The Replacements,2000, "Pain heals, Chicks dig scars... Glory lasts forever"

Unforgiven,1992, "It's a hell of a thing, killing a man"

The Birdcage,1996, Come as you are

The Replacements,2000, "Pain heals, Chicks dig scars... Glory lasts forever"

The Replacements,2000, "Pain heals, Chicks dig scars... Glory lasts forever"
The Birdcage, 1996, "Come as you are"
Unforgiven, 1992, "It's a hell of a thing, killing a man"
The Replacements, 2000, "Pain heals, Chicks dig scars... Glory lasts forever"
Cloud Atlas, 2012, "Everything is connected"
The Da Vinci Code, 2006, "Break The Codes"
The Replacements, 2000, "Pain heals, Chicks dig scars... Glory lasts forever"

Actors

The actors.csv file contains two columns title, roles, name and born.

The content of the actors.csv file:

<table>
<thead>
<tr>
<th>title</th>
<th>roles</th>
<th>name</th>
<th>born</th>
</tr>
</thead>
<tbody>
<tr>
<td>Something's Gotta Give</td>
<td>Julian Mercer</td>
<td>Keanu Reeves</td>
<td>1964</td>
</tr>
<tr>
<td>Johnny Mnemonic</td>
<td>Johnny Mnemonic</td>
<td>Keanu Reeves</td>
<td>1964</td>
</tr>
<tr>
<td>The Replacements</td>
<td>Shane Falco</td>
<td>Keanu Reeves</td>
<td>1964</td>
</tr>
<tr>
<td>The Devil's Advocate</td>
<td>Kevin Lomax</td>
<td>Keanu Reeves</td>
<td>1964</td>
</tr>
<tr>
<td>The Matrix Revolutions</td>
<td>Neo</td>
<td>Keanu Reeves</td>
<td>1964</td>
</tr>
<tr>
<td>The Matrix Reloaded</td>
<td>Neo</td>
<td>Keanu Reeves</td>
<td>1964</td>
</tr>
<tr>
<td>The Matrix</td>
<td>Neo</td>
<td>Keanu Reeves</td>
<td>1964</td>
</tr>
<tr>
<td>The Matrix Revolutions</td>
<td>Trinity, Carrie-Anne Moss</td>
<td>Keanu Reeves</td>
<td>1964</td>
</tr>
<tr>
<td>The Matrix Reloaded</td>
<td>Trinity, Carrie-Anne Moss</td>
<td>Keanu Reeves</td>
<td>1964</td>
</tr>
<tr>
<td>The Matrix Revolutions</td>
<td>Morpheus, Laurence Fishburne</td>
<td>Keanu Reeves</td>
<td>1964</td>
</tr>
<tr>
<td>The Matrix Reloaded</td>
<td>Morpheus, Laurence Fishburne</td>
<td>Keanu Reeves</td>
<td>1964</td>
</tr>
<tr>
<td>V for Vendetta</td>
<td>Hugo Weaving</td>
<td>Keanu Reeves</td>
<td>1964</td>
</tr>
<tr>
<td>Cloud Atlas, Bill Smoke</td>
<td>Haskell Moore, Tadeusz Kesselring, Nurse Noakes, Boardman Mephi, Old Georgie, Hugo Weaving, 1960</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The Matrix Revolutions</td>
<td>Agent Smith</td>
<td>Hugo Weaving, Keanu Reeves, 1964</td>
<td></td>
</tr>
<tr>
<td>The Matrix Reloaded</td>
<td>Agent Smith</td>
<td>Hugo Weaving, Keanu Reeves, 1964</td>
<td></td>
</tr>
<tr>
<td>The Matrix</td>
<td>Agent Smith</td>
<td>Hugo Weaving, Keanu Reeves, 1964</td>
<td></td>
</tr>
<tr>
<td>The Matrix</td>
<td>Emil</td>
<td>Hugo Weaving, Keanu Reeves, 1964</td>
<td></td>
</tr>
<tr>
<td>That Thing You Do</td>
<td>Tina</td>
<td>Charlize Theron, 1975</td>
<td></td>
</tr>
<tr>
<td>The Devil's Advocate</td>
<td>Mary Ann Lomax</td>
<td>Charlize Theron, 1975</td>
<td></td>
</tr>
<tr>
<td>The Devil's Advocate</td>
<td>John Milton</td>
<td>Al Pacino, 1940</td>
<td></td>
</tr>
<tr>
<td>Jerry Maguire</td>
<td>Jerry Maguire</td>
<td>Tom Cruise, 1962</td>
<td></td>
</tr>
<tr>
<td>Top Gun</td>
<td>Maverick</td>
<td>Tom Cruise, 1962</td>
<td></td>
</tr>
<tr>
<td>A Few Good Men</td>
<td>Lt. Daniel Kaffee</td>
<td>Tom Cruise, 1962</td>
<td></td>
</tr>
<tr>
<td>Something's Gotta Give</td>
<td>Harry Sanborn</td>
<td>Jack Nicholson, 1937</td>
<td></td>
</tr>
<tr>
<td>One Flew Over the Cuckoo's Nest</td>
<td>Randle McMurphy</td>
<td>Jack Nicholson, 1937</td>
<td></td>
</tr>
<tr>
<td>As Good as It Gets</td>
<td>Melvin Udall</td>
<td>Jack Nicholson, 1937</td>
<td></td>
</tr>
<tr>
<td>Apollo 13</td>
<td>Jack Swigert</td>
<td>Kevin Bacon, 1958</td>
<td></td>
</tr>
<tr>
<td>Frost/Nixon</td>
<td>Jack Brennan</td>
<td>Kevin Bacon, 1958</td>
<td></td>
</tr>
<tr>
<td>Stand By Me</td>
<td>Ace Merrill</td>
<td>Kiefer Sutherland, 1966</td>
<td></td>
</tr>
<tr>
<td>A Few Good Men</td>
<td>Lt. Jonathan Kendrick</td>
<td>Kiefer Sutherland, 1966</td>
<td></td>
</tr>
<tr>
<td>What Dreams May Come</td>
<td>Albert Lewis, Cuba Gooding Jr., 1968</td>
<td></td>
<td></td>
</tr>
<tr>
<td>As Good as It Gets</td>
<td>Frank Sachs, Cuba Gooding Jr., 1968</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jerry Maguire</td>
<td>Rod Tidwell, Cuba Gooding Jr., 1968</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A Few Good Men</td>
<td>Lt. Sam Weinberg</td>
<td>Kevin Pollak, 1957</td>
<td></td>
</tr>
<tr>
<td>Hoffa, Frank Fitzsimmons, J.T. Walsh, 1943</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A Few Good Men</td>
<td>Lt. Col. Matthew Andrew Markinson, J.T. Walsh, 1943</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A Few Good Men</td>
<td>Dr. Stone</td>
<td>Christopher Guest, 1948</td>
<td></td>
</tr>
<tr>
<td>A Few Good Men</td>
<td>Man in Bar</td>
<td>Aaron Sorkin, 1961</td>
<td></td>
</tr>
<tr>
<td>Top Gun</td>
<td>Charlie, Kelly McGillis</td>
<td>1957</td>
<td></td>
</tr>
<tr>
<td>Top Gun</td>
<td>Iceman, Val Kilmer</td>
<td>1959</td>
<td></td>
</tr>
<tr>
<td>Top Gun</td>
<td>Goose, Anthony Edwards</td>
<td>1962</td>
<td></td>
</tr>
<tr>
<td>Top Gun</td>
<td>Viper, Tom Skerritt</td>
<td>1933</td>
<td></td>
</tr>
<tr>
<td>When Harry Met Sally</td>
<td>Sally Albright</td>
<td>Meg Ryan, 1961</td>
<td></td>
</tr>
<tr>
<td>Joe Versus the Volcano</td>
<td>DeDe</td>
<td>Angelica Graynamore, Patricia Graynamore, Meg Ryan, 1961</td>
<td></td>
</tr>
<tr>
<td>Sleepless in Seattle</td>
<td>Annie Reed</td>
<td>Meg Ryan, 1961</td>
<td></td>
</tr>
<tr>
<td>You've Got Mail</td>
<td>Kathleen Kelly</td>
<td>Meg Ryan, 1961</td>
<td></td>
</tr>
<tr>
<td>Title</td>
<td>Character</td>
<td>Actor</td>
<td>Year</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------</td>
<td>------------</td>
<td>------</td>
</tr>
<tr>
<td>The Da Vinci Code</td>
<td>Silas</td>
<td>Paul Bettany</td>
<td>1971</td>
</tr>
<tr>
<td>V for Vendetta</td>
<td>Evey Hammond</td>
<td>Natalie Portman</td>
<td>1981</td>
</tr>
<tr>
<td>V for Vendetta</td>
<td>Eric Finch</td>
<td>Stephen Rea</td>
<td>1946</td>
</tr>
<tr>
<td>V for Vendetta</td>
<td>High Chancellor</td>
<td>Adam Sutler, John Hurt</td>
<td>1940</td>
</tr>
<tr>
<td>Ninja Assassin</td>
<td>Ryan Maslow</td>
<td>Ben Miles</td>
<td>1967</td>
</tr>
<tr>
<td>Speed Racer</td>
<td>Cass Jones</td>
<td>Ben Miles</td>
<td>1967</td>
</tr>
<tr>
<td>V for Vendetta</td>
<td>Dascomb</td>
<td>Ben Miles</td>
<td>1967</td>
</tr>
<tr>
<td>Speed Racer</td>
<td>Speed Racer</td>
<td>Emile Hirsch</td>
<td>1985</td>
</tr>
<tr>
<td>Speed Racer</td>
<td>Pops</td>
<td>John Goodman</td>
<td>1960</td>
</tr>
<tr>
<td>Speed Racer</td>
<td>Mom</td>
<td>Susan Sarandon</td>
<td>1946</td>
</tr>
<tr>
<td>Speed Racer</td>
<td>Racer X</td>
<td>Matthew Fox</td>
<td>1966</td>
</tr>
<tr>
<td>Speed Racer</td>
<td>Trixie</td>
<td>Christina Ricci</td>
<td>1980</td>
</tr>
<tr>
<td>Ninja Assassin</td>
<td>Raizo</td>
<td>Rain</td>
<td>1982</td>
</tr>
<tr>
<td>Speed Racer</td>
<td>Taejo Togokahn</td>
<td>Rain</td>
<td>1982</td>
</tr>
<tr>
<td>Ninja Assassin</td>
<td>Mika Coretti</td>
<td>Naomie Harris</td>
<td>null</td>
</tr>
<tr>
<td>The Green Mile</td>
<td>John Coffey</td>
<td>Michael Clarke Duncan</td>
<td>1957</td>
</tr>
<tr>
<td>The Green Mile</td>
<td>Brutus 'Brutal' Howell</td>
<td>David Morse</td>
<td>1953</td>
</tr>
<tr>
<td>Frost/Nixon</td>
<td>&quot;James Reston, Jr.&quot;</td>
<td>Sam Rockwell</td>
<td>1968</td>
</tr>
<tr>
<td>The Green Mile</td>
<td>'Wild Bill' Wharton</td>
<td>Sam Rockwell</td>
<td>1968</td>
</tr>
<tr>
<td>Apollo 13</td>
<td>Ken Mattingly</td>
<td>Gary Sinise</td>
<td>1955</td>
</tr>
<tr>
<td>The Green Mile</td>
<td>Burt Hammersmith</td>
<td>Gary Sinise</td>
<td>1955</td>
</tr>
<tr>
<td>The Green Mile</td>
<td>Melinda Moores</td>
<td>Patricia Clarkson</td>
<td>1959</td>
</tr>
<tr>
<td>Frost/Nixon</td>
<td>Richard Nixon</td>
<td>Frank Langella</td>
<td>1938</td>
</tr>
<tr>
<td>Frost/Nixon</td>
<td>David Frost</td>
<td>Michael Sheen</td>
<td>1969</td>
</tr>
<tr>
<td>Bicentennial Man</td>
<td>Rupert Burns</td>
<td>Oliver Platt</td>
<td>1960</td>
</tr>
<tr>
<td>Frost/Nixon</td>
<td>Bob Zelnick</td>
<td>Oliver Platt</td>
<td>1960</td>
</tr>
<tr>
<td>One Flew Over the Cuckoo's Nest</td>
<td>Martini</td>
<td>Danny DeVito</td>
<td>1944</td>
</tr>
<tr>
<td>Hoffa, Robert 'Bobby' Ciaron</td>
<td>Danny DeVito</td>
<td>1944</td>
<td></td>
</tr>
<tr>
<td>Hoffa, Peter 'Pete' Connelly</td>
<td>John C. Reilly</td>
<td>1965</td>
<td></td>
</tr>
<tr>
<td>Apollo 13</td>
<td>Gene Kranz</td>
<td>Ed Harris</td>
<td>1950</td>
</tr>
<tr>
<td>A League of Their Own</td>
<td>Bob Hinson</td>
<td>Bill Paxton</td>
<td>1955</td>
</tr>
<tr>
<td>Twister</td>
<td>Bill Harding</td>
<td>Bill Paxton</td>
<td>1955</td>
</tr>
<tr>
<td>Apollo 13</td>
<td>Fred Haise</td>
<td>Bill Paxton</td>
<td>1955</td>
</tr>
<tr>
<td>Charlie Wilson's War</td>
<td>Gust Avrakotos</td>
<td>Philip Seymour Hoffman</td>
<td>1967</td>
</tr>
<tr>
<td>Twister, Dustin 'Dusty' Davis</td>
<td>Philip Seymour Hoffman</td>
<td>1967</td>
<td></td>
</tr>
<tr>
<td>Something's Gotta Give</td>
<td>Erica Barry</td>
<td>Diane Keaton</td>
<td>1946</td>
</tr>
<tr>
<td>Charlie Wilson's War</td>
<td>Joanne Herring</td>
<td>Julia Roberts</td>
<td>1967</td>
</tr>
<tr>
<td>A League of Their Own</td>
<td>'All the Way' Mae Mordabito</td>
<td>Madonna</td>
<td>1954</td>
</tr>
<tr>
<td>A League of Their Own</td>
<td>Dottie Hinson</td>
<td>Geena Davis</td>
<td>1956</td>
</tr>
<tr>
<td>A League of Their Own</td>
<td>Kit Keller</td>
<td>Lori Petty</td>
<td>1963</td>
</tr>
</tbody>
</table>

**Directors**

The `directors.csv` file contains two columns *title*, *name* and *born*.

The content of the `directors.csv` file:
28.5.2. Prerequisites

The example uses the Linux or macOS tarball installation. It assumes that your current work directory is the `<neo4j-home>` directory of the tarball installation, and the CSV files are placed in the default import directory.

- For the default directory of other installations see, [Operations Manual → File locations](#).

- The import location can be configured with [Operations Manual → dbms.directories.import](#).

28.5.3. Importing the data

Import the movies.csv file
LOAD CSV WITH HEADERS FROM 'file:///movies.csv' AS line
MERGE (m:Movie {title: line.title})
ON CREATE SET
  m.released = toInteger(line.released),
  m.tagline = line.tagline

Added 38 nodes, Set 114 properties, Added 38 labels

**Import the actors.csv file**

LOAD CSV WITH HEADERS FROM 'file:///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)

Added 102 nodes, Created 172 relationships, Set 375 properties, Added 102 labels

**Import the directors.csv file**

LOAD CSV WITH HEADERS FROM 'file:///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)

Added 23 nodes, Created 44 relationships, Set 46 properties, Added 23 labels

**Create an index for nodes with the Person label**

CREATE INDEX FOR (p:Person)
ON (p.name)

Added 1 indexes

CALL db.awaitIndexes

**28.5.4. Index-backed property-lookup**

In this example you want to write a query to find persons with the name 'Tom' that acted in a movie.

MATCH (p:Person)-[:ACTED_IN]-(m:Movie)
WHERE p.name STARTS WITH 'Tom'
RETURN
  p.name AS name,
  count(m) AS count
The query requests the database to return all the actors with the first name 'Tom'. There are three of them: 'Tom Cruise', 'Tom Skerritt' and 'Tom Hanks'. With native indexes, however, you can leverage the fact that indexes store the property values. In this case, it means that the names can be looked up directly from the index. This allows Cypher to avoid the second call to the database to find the property, which can save time on very large queries.

If we profile the above query, we see that the `NodeIndexSeekByRange` in the `Details` column contains `cache[p.name]`, which means that `p.name` is retrieved from the index. We can also see that the `OrderedAggregation` has no `DB Hits`, which means it does not have to access the database again.

```cypher
PROFILE
MATCH (p:Person)-[r:ACTED_IN]->(m:Movie)
WHERE p.name STARTS WITH 'Tom'
RETURN
  p.name AS name,
  count(m) AS count
```
If we change the query, such that it can no longer use an index, we will see that there will be no `cache[p.name]` in the Details column, and that the EagerAggregation now has DB Hits, since it accesses the database again to retrieve the name.

```sql
PROFILE
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
RETURN p.name AS name, count(m) AS count
```

<table>
<thead>
<tr>
<th>name</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Diane Keaton&quot;</td>
<td>1</td>
</tr>
<tr>
<td>Actor</td>
<td>Count</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>Jack Nicholson</td>
<td>5</td>
</tr>
<tr>
<td>Keanu Reeves</td>
<td>7</td>
</tr>
<tr>
<td>Ice-T</td>
<td>1</td>
</tr>
<tr>
<td>Takeshi Kitano</td>
<td>1</td>
</tr>
<tr>
<td>Dina Meyer</td>
<td>1</td>
</tr>
<tr>
<td>Brooke Langton</td>
<td>1</td>
</tr>
<tr>
<td>Gene Hackman</td>
<td>3</td>
</tr>
<tr>
<td>Orlando Jones</td>
<td>1</td>
</tr>
<tr>
<td>Al Pacino</td>
<td>1</td>
</tr>
<tr>
<td>Charlene Tilton</td>
<td>2</td>
</tr>
<tr>
<td>Hugo Weaving</td>
<td>5</td>
</tr>
<tr>
<td>Laurence Fishburne</td>
<td>3</td>
</tr>
<tr>
<td>Carrie-Anne Moss</td>
<td>3</td>
</tr>
<tr>
<td>Emil Eifrem</td>
<td>1</td>
</tr>
<tr>
<td>John Hurt</td>
<td>1</td>
</tr>
<tr>
<td>Stephen Rea</td>
<td>1</td>
</tr>
<tr>
<td>Natalie Portman</td>
<td>1</td>
</tr>
<tr>
<td>Ben Miles</td>
<td>3</td>
</tr>
<tr>
<td>Jim Broadbent</td>
<td>1</td>
</tr>
<tr>
<td>Tom Hanks</td>
<td>12</td>
</tr>
<tr>
<td>Halle Berry</td>
<td>1</td>
</tr>
<tr>
<td>John Goodman</td>
<td>1</td>
</tr>
<tr>
<td>Susan Sarandon</td>
<td>1</td>
</tr>
<tr>
<td>Christina Ricci</td>
<td>1</td>
</tr>
<tr>
<td>Rain</td>
<td>2</td>
</tr>
<tr>
<td>Emile Hirsch</td>
<td>1</td>
</tr>
<tr>
<td>Matthew Fox</td>
<td>1</td>
</tr>
<tr>
<td>Rick Yune</td>
<td>2</td>
</tr>
<tr>
<td>Naomi Harris</td>
<td>1</td>
</tr>
<tr>
<td>Liv Tyler</td>
<td>1</td>
</tr>
<tr>
<td>Kelly Preston</td>
<td>1</td>
</tr>
<tr>
<td>Bonnie Hunt</td>
<td>2</td>
</tr>
<tr>
<td>Jerry O'Connell</td>
<td>2</td>
</tr>
<tr>
<td>Renee Zellweger</td>
<td>1</td>
</tr>
<tr>
<td>Jay Mohr</td>
<td>1</td>
</tr>
<tr>
<td>Jonathan Lipnicki</td>
<td>1</td>
</tr>
<tr>
<td>Cuba Gooding Jr.</td>
<td>4</td>
</tr>
<tr>
<td>Regina King</td>
<td>1</td>
</tr>
<tr>
<td>Tom Cruise</td>
<td>3</td>
</tr>
<tr>
<td>Kelly McGillis</td>
<td>1</td>
</tr>
<tr>
<td>Anthony Edwards</td>
<td>1</td>
</tr>
<tr>
<td>Tom Skerritt</td>
<td>1</td>
</tr>
<tr>
<td>Meg Ryan</td>
<td>5</td>
</tr>
<tr>
<td>Val Kilmer</td>
<td>1</td>
</tr>
<tr>
<td>Kiefer Sutherland</td>
<td>2</td>
</tr>
<tr>
<td>Kevin Bacon</td>
<td>3</td>
</tr>
<tr>
<td>Aaron Sorkin</td>
<td>1</td>
</tr>
<tr>
<td>Christopher Guest</td>
<td>1</td>
</tr>
<tr>
<td>Noah Wyle</td>
<td>1</td>
</tr>
<tr>
<td>James Marshall</td>
<td>1</td>
</tr>
<tr>
<td>Kevin Pollak</td>
<td>1</td>
</tr>
<tr>
<td>J.T. Walsh</td>
<td>2</td>
</tr>
<tr>
<td>Demi Moore</td>
<td>1</td>
</tr>
<tr>
<td>Danny DeVito</td>
<td>2</td>
</tr>
<tr>
<td>John C. Reilly</td>
<td>1</td>
</tr>
<tr>
<td>Helen Hunt</td>
<td>3</td>
</tr>
<tr>
<td>Greg Kinnear</td>
<td>2</td>
</tr>
<tr>
<td>Ed Harris</td>
<td>1</td>
</tr>
<tr>
<td>Bill Paxton</td>
<td>3</td>
</tr>
<tr>
<td>Gary Sinise</td>
<td>2</td>
</tr>
<tr>
<td>Oliver Platt</td>
<td>2</td>
</tr>
<tr>
<td>Frank Langella</td>
<td>1</td>
</tr>
<tr>
<td>Michael Sheen</td>
<td>1</td>
</tr>
<tr>
<td>Sam Rockwell</td>
<td>2</td>
</tr>
<tr>
<td>John Cusack</td>
<td>1</td>
</tr>
<tr>
<td>Wil Wheaton</td>
<td>1</td>
</tr>
<tr>
<td>Corey Feldman</td>
<td>1</td>
</tr>
<tr>
<td>River Phoenix</td>
<td>1</td>
</tr>
<tr>
<td>Marshall Bell</td>
<td>2</td>
</tr>
<tr>
<td>Max von Sydow</td>
<td>2</td>
</tr>
<tr>
<td>Annabella Sciorra</td>
<td>1</td>
</tr>
<tr>
<td>Werner Herzog</td>
<td>1</td>
</tr>
<tr>
<td>Robin Williams</td>
<td>3</td>
</tr>
<tr>
<td>Billy Crystal</td>
<td>1</td>
</tr>
<tr>
<td>Carrie Fisher</td>
<td>1</td>
</tr>
<tr>
<td>Bruno Kirby</td>
<td>1</td>
</tr>
<tr>
<td>Nathan Lane</td>
<td>2</td>
</tr>
<tr>
<td>Plan</td>
<td>Statement</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>&quot;PROFILE&quot;</td>
<td>&quot;READ_ONLY&quot;</td>
</tr>
</tbody>
</table>

For non-native indexes there will still be a second database access to retrieve those values.

Predicates that can be used to enable this optimization are:

- **Existence** (e.g. `WHERE n.name IS NOT NULL`)
- **Equality** (e.g. `WHERE n.name = 'Tom Hanks'`)
- **Range** (e.g. `WHERE n.uid > 1000 AND n.uid < 2000`)
- **Prefix** (e.g. `WHERE n.name STARTS WITH 'Tom'`)

102 rows
• Suffix (e.g. WHERE n.name ENDS WITH 'Hanks')
• Substring (e.g. WHERE n.name CONTAINS 'a')
• Several predicates of the above types combined using OR, given that all of them are on the same property (e.g. WHERE n.prop < 10 OR n.prop = 'infinity')

If there is an existence constraint on the property, no predicate is required to trigger the optimization. For example, CREATE CONSTRAINT constraint_name FOR (p:Person) REQUIRE p.name IS NOT NULL.

Aggregating functions

For all built-in aggregating functions in Cypher, the index-backed property-lookup optimization can be used even without a predicate.

Consider this query which returns the number of distinct names of people in the movies dataset:

```cypher
PROFILE
MATCH (p:Person)
RETURN count(DISTINCT p.name) AS numberOfNames
```

<table>
<thead>
<tr>
<th>numberOfNames</th>
</tr>
</thead>
<tbody>
<tr>
<td>125</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Plan</th>
<th>Statement</th>
<th>Version</th>
<th>Planner</th>
<th>Runtime</th>
<th>Time</th>
<th>DbHits</th>
<th>Rows</th>
<th>Memory (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;PROFILE&quot;</td>
<td>&quot;READ_ONLY&quot;</td>
<td>&quot;CYPHER 4.3&quot;</td>
<td>&quot;COST&quot;</td>
<td>&quot;PIPELINED&quot;</td>
<td>45</td>
<td>126</td>
<td>1</td>
<td>9952</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
<th>Estimated Rows</th>
<th>Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults@neo4j</td>
<td>numberOfNames</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>0.048</td>
<td>In Pipeline 1</td>
</tr>
<tr>
<td>+EagerAggregation@neo4j</td>
<td>count(DISTINCT cache[p.name]) AS numberOfNames</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>9888</td>
<td></td>
<td>Fused in Pipeline 0</td>
</tr>
<tr>
<td>+NodeIndexScan@neo4j</td>
<td>p:Person(name) WHERE name IS NOT NULL, cache[p.name]</td>
<td>125</td>
<td>125</td>
</tr>
<tr>
<td>126</td>
<td>72</td>
<td>1/0</td>
<td>1.569</td>
</tr>
</tbody>
</table>

1 row

Note that the NodeIndexScan in the Details column contains cache[p.name] and that the EagerAggregation has no DB Hits. In this case, the semantics of aggregating functions works like an implicit existence predicate because Person nodes without the property name will not affect the result of an aggregation.
28.5.5. Index-backed order by

Now consider the following refinement to the query:

```cyphe
PROFILE
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WHERE p.name STARTS WITH 'Tom'
RETURN
  p.name AS name,
  count(m) AS count
ORDER BY name
```

<table>
<thead>
<tr>
<th>name</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Tom Cruise&quot;</td>
<td>3</td>
</tr>
<tr>
<td>&quot;Tom Hanks&quot;</td>
<td>12</td>
</tr>
<tr>
<td>&quot;Tom Skerritt&quot;</td>
<td>1</td>
</tr>
</tbody>
</table>

We are asking for the results in ascending alphabetical order. The native index happens to store String
properties in ascending alphabetical order, and Cypher knows this. In Neo4j 3.5 and later, the Cypher planner will recognize that the index already returns data in the correct order, and skip the Sort operation.

The Order by column describes the order of rows after each operator. We see that the Order by column contains p.name ASC from the index seek operation, meaning that the rows are ordered by p.name in ascending order.

Index-backed order by can also be used for queries that expect their results is descending order, but with slightly lower performance.

In cases where the Cypher planner is unable to remove the Sort operator, the planner can utilize knowledge of the ORDER BY clause to plan the Sort operator at a point in the plan with optimal cardinality.

**min() and max()**

For the min and max functions, the index-backed order by optimization can be used to avoid aggregation and instead utilize the fact that the minimum/maximum value is the first/last one in a sorted index. Consider the following query which returns the fist actor in alphabetical order:

```cypher
PROFILE
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
RETURN min(p.name) AS name
```
### Plan

<table>
<thead>
<tr>
<th>Plan</th>
<th>Statement</th>
<th>Version</th>
<th>Planner</th>
<th>Runtime</th>
<th>Time</th>
<th>DbHits</th>
<th>Rows</th>
<th>Memory (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;PROFILE&quot;</td>
<td>&quot;READ_ONLY&quot;</td>
<td>&quot;CYPER 4.3&quot;</td>
<td>&quot;COST&quot;</td>
<td>&quot;PIPELINED&quot;</td>
<td>38</td>
<td>809</td>
<td>1</td>
<td>184</td>
</tr>
</tbody>
</table>

### Operator Details

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Memory (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults@neo4j</td>
<td>name</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0/0</td>
<td>0.041</td>
<td>In Pipeline 1</td>
<td></td>
</tr>
<tr>
<td>+EagerAggregation@neo4j</td>
<td>min(p.name) AS name</td>
<td>1</td>
<td>1</td>
<td>344</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fused in Pipeline 0</td>
<td></td>
</tr>
<tr>
<td>+Filter@neo4j</td>
<td>p:Person</td>
<td>172</td>
<td>172</td>
<td>172</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fused in Pipeline 0</td>
<td></td>
</tr>
<tr>
<td>+Expand(All)@neo4j</td>
<td>(m)&lt;-[anon_16:ACTED_IN]-(p)</td>
<td>172</td>
<td>172</td>
<td>254</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fused in Pipeline 0</td>
<td></td>
</tr>
<tr>
<td>+NodeByLabelScan@neo4j</td>
<td>m:Movie</td>
<td>38</td>
<td>38</td>
<td>39</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5/0</td>
<td>1.636</td>
<td>Fused in Pipeline 0</td>
<td></td>
</tr>
</tbody>
</table>

1 row

Aggregations are usually using the **EagerAggregation** operation. This would mean scanning all nodes in the index to find the name that is first in alphabetic order. Instead, the query is planned with **Projection**, followed by **Limit**, followed by **Optional**. This will simply pick the first value from the index.

For large datasets, this can improve performance dramatically.

**Index-backed order by** can also be used for corresponding queries with the **max** function, but with slightly lower performance.

### Restrictions

The optimization can only work on native indexes. It does not work for predicates only querying for the spatial type **Point**.

Predicates that can be used to enable this optimization are:

- **Existence** (e.g. `WHERE n.name IS NOT NULL`)
- **Equality** (e.g. `WHERE n.name = 'Tom Hanks'`)
- **Range** (e.g. `WHERE n.uid > 1000 AND n.uid < 2000`)
- **Prefix** (e.g. `WHERE n.name STARTS WITH 'Tom'`)

---

611
• Suffix (e.g. WHERE n.name ENDS WITH 'Hanks')
• Substring (e.g. WHERE n.name CONTAINS 'a')

Predicates that will not work:

• Several predicates combined using OR
• Equality or range predicates querying for points (e.g. WHERE n.place > point({ x: 1, y: 2 }))
• Spatial distance predicates (e.g. WHERE point.distance(n.place, point({ x: 1, y: 2 })) < 2)

If there is an existence constraint on the property, no predicate is required to trigger the optimization. For example,

CREATE CONSTRAINT constraint_name FOR (p:Person)
REQUIRE p.name IS NOT NULL

As of Neo4j 4.4.10, predicates with parameters, such as WHERE n.prop > $param, can trigger index-backed order by. The only exception are queries with parameters of type Point.

28.6. 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

28.6.1. 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 suboptimal 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.
COMMIT query hint.

Query

```
MATCH (s:Scientist {born: 1850})-[RESEARCHED]->
  (sc:Science)<-[INVENTED_BY {year: 560}]-(
  p:Pioneer {born: 525})-[LIVES_IN]->
  (c:City)<-[PART_OF]->
  (cc:Country {formed: 411})

RETURN *
```

The query above will be used in some of the examples on this page. Without any hints, one index and no join is used.

Query plan

```
Compiler CYPHER 4.4
Planner COST
Runtime PIPELINED
Runtime version 4.4

+-----------------------------------+-----------------------------------+-----------------------------------+-----------------------------------+---------------------------------------------+
| Operator                          | Details                           | Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Other |
|-----------------------------------+-----------------------------------+-----------------------------------+-----------------------------------+---------------------------------------------+
| +ProduceResults                   | c, cc, i, p, s, sc                |              0 |    0 |        |                |                        |           | Fused in Pipeline 0 |
| |                                  |                                   |           | Fused in Pipeline 0 |
| +Filter                           | s.born = $autoint_0 AND s:Scientist |              0 |    0 |        |                |                        |           | Fused in Pipeline 0 |
| |                                  |                                   |           | Fused in Pipeline 0 |
| +Expand(All)                      | (sc)<-[anon_0:RESEARCHED]-(s)     |              0 |    0 |        |                |                        |           | Fused in Pipeline 0 |
| |                                  |                                   |           | Fused in Pipeline 0 |
| +Filter                           | i.year = $autoint_1 AND sc:Science |              0 |    0 |        |                |                        |           | Fused in Pipeline 0 |
| |                                  |                                   |           | Fused in Pipeline 0 |
| +Expand(All)                      | (p)-[i:INVENTED_BY]->(sc)         |              0 |    0 |        |                |                        |           | Fused in Pipeline 0 |
| |                                  |                                   |           | Fused in Pipeline 0 |
| +Filter                           | p.born = $autoint_2 AND p:Pioneer |              0 |    0 |        |                |                        |           | Fused in Pipeline 0 |
| |                                  |                                   |           | Fused in Pipeline 0 |
| +Expand(All)                      | (c)<-[anon_1:LIVES_IN]-(p)        |              1 |    1 |        |                |                        |           | Fused in Pipeline 0 |
| |                                  |                                   |           | Fused in Pipeline 0 |
| +Filter                           | c:City                            |              1 |    1 |        |                |                        |           | Fused in Pipeline 0 |
| |                                  |                                   |           | Fused in Pipeline 0 |
| +Expand(All)                      | (cc)<-[anon_2:PART_OF]-(c)        |              1 |    1 |        |                |                        |           | Fused in Pipeline 0 |
| |                                  |                                   |           | Fused in Pipeline 0 |
| +NodeIndexSeek                    | BTREE INDEX cc:Country(formed) WHERE formed = $autoint_3 |              1 |    1 |        |                |                        |           | Fused in Pipeline 0 |

Total database accesses: 10, total allocated memory: 200
28.6.2. Index hints

Index hints are used to specify which index 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. An index hint is supplied after an applicable MATCH clause. Available index hints are:

<table>
<thead>
<tr>
<th>Hint</th>
<th>Fulfilled by plans</th>
</tr>
</thead>
<tbody>
<tr>
<td>USING [BTREE</td>
<td>TEXT] INDEX variable:Label(property)</td>
</tr>
<tr>
<td>USING [BTREE</td>
<td>TEXT] INDEX SEEK variable:Label(property)</td>
</tr>
<tr>
<td>USING [BTREE</td>
<td>TEXT] INDEX variable:RELATIONSHIP_TYPE(property)</td>
</tr>
<tr>
<td>USING [BTREE</td>
<td>TEXT] INDEX SEEK variable:RELATIONSHIP_TYPE(property)</td>
</tr>
</tbody>
</table>

When specifying an index type for a hint, e.g. BTREE or TEXT, the hint can only be fulfilled when an index of the specified type is available. When no index type is specified, the hint can be fulfilled by any index types.

Using a hint must never change the result of a query. Therefore, a hint with a specified index type is only fulfillable when the planner knows that using an index of the specified type does not change the results. Please refer to The use of indexes for more details.

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 a node index hint

The query above can be tuned to pick a different index as the starting point.

**Query**

```
MATCH (s:Scientist {born: 1850})-[[:RESEARCHED]]->
(sc:Science)<-[[:INVENTED_BY] {year: 560}]->
(p:Pioneer {born: 525})-[[:LIVES_IN]]->
(c:City)-[:PART_OF]->
(cc:Country {formed: 411})
USING INDEX p:Pioneer(born)
RETURN *
```
Query plan

Compiler CYPHER 4.4
Planner COST
Runtime PIPELINED
Runtime version 4.4

| Operator        | Details                                             | Estimated Rows | Rows | DB Hits |
|-----------------+-----------------------------------------------------+----------------+------+---------|
| Memory (Bytes)  | Page Cache Hits/Misses | Time (ms) | Other               |
|-----------------+-----------------------------------------------------+----------------+------+---------|
| +ProduceResults | c, cc, i, p, s, sc                                  |              0 |    0 |       0 |
|                | Fused in Pipeline 0                                 |               |      |         |
| +Filter         | cc.formed = $autoint_3 AND cc:Country               |              0 |    0 |       0 |
|                | Fused in Pipeline 0                                 |               |      |         |
| +Expand(All)    | (c)-[anon_2:PART_OF]->(cc)                          |              0 |    0 |       0 |
|                | Fused in Pipeline 0                                 |               |      |         |
| +Filter         | c:City                                              |              0 |    0 |       0 |
|                | Fused in Pipeline 0                                 |               |      |         |
| +Expand(All)    | (p)-[anon_1:LIVES_IN]->(c)                          |              0 |    0 |       0 |
|                | Fused in Pipeline 0                                 |               |      |         |
| +Filter         | s.born = $autoint_0 AND s:Scientist                 |              0 |    0 |       0 |
|                | Fused in Pipeline 0                                 |               |      |         |
| +Expand(All)    | (sc)<-[anon_0:RESEARCHED]-(s)                       |              0 |    0 |       0 |
|                | Fused in Pipeline 0                                 |               |      |         |
| +Filter         | i.year = $autoint_1 AND sc:Science                  |              0 |    0 |       2 |
|                | Fused in Pipeline 0                                 |               |      |         |
| +Expand(All)    | (p)-[i:INVENTED_BY]->(sc)                           |              2 |    2 |       6 |
|                | Fused in Pipeline 0                                 |               |      |         |
| +NodeIndexSeek  | BTREE INDEX p:Pioneer(born) WHERE born = $autoint_2 |              2 |    2 |       3 |
|                | 112 | 4/1 | 0.790 | Fused in Pipeline 0 |

Total database accesses: 11, total allocated memory: 200

Query using a node text index hint

The following query can be tuned to pick a text index.

```
MATCH (c:Country)
USING TEXT INDEX c:Country(name)
WHERE c.name = 'Country7'
RETURN *
```
Query plan

Compiler CYPHER 4.4
Planner COST
Runtime PIPELINED
Runtime version 4.4

+-----------------+-------------------------------------------------------+----------------+------+
| Operator        | Details                                               | Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Other               |
+-----------------+-------------------------------------------------------+----------------+------+
| +ProduceResults | c                                                     |              1 |    1 | Fused in Pipeline 0 |

+---------+----------------+------------------------+-----------+---------------------+
| +NodeIndexSeek | TEXT INDEX c:Country(name) WHERE name = $autostring_0 |              1 |    1 | 31.274 | Fused in Pipeline 0 |

Total database accesses: 2, total allocated memory: 176

Query using a relationship index hint

The query above can be tuned to pick a relationship index as the starting point.

Query

MATCH (s:Scientist {born: 1850})-[[:RESEARCHED]]->
(sc:Science)<-[[:INVENTED_BY]]{{year: 560}}->
(p:Pioneer {born: 525})-[[:LIVES_IN]]->
(c:City){:PART_OF}->
(cc:Country {formed: 411})

USING INDEX i:INVENTED_BY(year)

RETURN *
### Query plan

**Compiler CYpher 4.4**

**Planner COST**

**Runtime PIPELINED**

**Runtime version 4.4**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimated Rows</td>
<td>Rows</td>
</tr>
<tr>
<td>+ProduceResults</td>
<td>c, cc, i, p, s, sc</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+Filter</td>
<td>cc.formed = $autoint_3 AND cc:Country</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+Expand(All)</td>
<td>(c)-[anon_2:PART_OF]-&gt;(cc)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+Filter</td>
<td>c:City</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+Expand(All)</td>
<td>(p)-[anon_1:LIVES_IN]-&gt;(c)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+Filter</td>
<td>s.born = $autoint_0 AND s:Scientist</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+Expand(All)</td>
<td>(sc)&lt;-[anon_0:RESEARCHED]-(s)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+Filter</td>
<td>p.born = $autoint_2 AND sc:Science AND p:Pioneer</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+DirectedRelationshipIndexSeek</td>
<td>BTREE INDEX (p)-[i:INVENTED_BY(year)]-&gt;(sc) WHERE year = $autoint_1</td>
</tr>
<tr>
<td>+----------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>

Total database accesses: 9, total allocated memory: 200

---

**Query using a relationship text index hint**

The following query can be tuned to pick a text index.
Query

```
MATCH ()-[i:INVENTED_BY]->()
USING TEXT INDEX i:INVENTED_BY(location)
WHERE i.location = 'Location7'
RETURN *
```

Query plan

```
Compiler CYPHER 4.4
Planner COST
Runtime PIPELINED
Runtime version 4.4

+-------------------------------------+
<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>i</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>+DirectedRelationshipIndexSeek</td>
<td>TEXT INDEX (anon_0)-[i:INVENTED_BY(location)]-&gt;(anon_1) WHERE location = $autostring_0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
+-------------------------------------+
```

Total database accesses: 3, total allocated memory: 176

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 (s:Scientist {born: 1850})-[[:RESEARCHED]]-> (sc:Science)<-[i:INVENTED_BY {year: 560}]- (p:Pioneer {born: 525})-[[:LIVES_IN]]-> (c:City)[-[:PART_OF]-> (cc:Country {formed: 411}))
USING INDEX s:Scientist(born)
USING INDEX cc:Country(formed)
RETURN *
```
### Query plan

Compiler CYPHER 4.4  
Planner COST  
Runtime PIPELINED  
Runtime version 4.4

| Operator          | Details                                                      | Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Other               |
|-------------------|--------------------------------------------------------------|----------------+------+---------+---------------+------------------------+-----------+---------------------|
| +ProduceResults   | c, cc, i, p, s, sc                                           |               0 |     0 |        0 |               |                        |     0.000 | In Pipeline 2       |
| +NodeHashJoin     | sc                                                           |               0 |     0 |        0 |       432      |                        |           | In Pipeline 2       |
| +Expand(All)      | (s)-[anon_0:RESEARCHED]->(sc)                               |               1 |     0 |        0 |               |                        |           | Fused in Pipeline 1 |
| +NodeIndexSeek    | BTREE INDEX s:Scientist(born) WHERE born = $autoint_0       |               1 |     0 |        0 |       112      |                        |     0.000 | Fused in Pipeline 1 |
| +Filter           | i.year = $autoint_1 AND sc:Science                           |               0 |     0 |        0 |               |                        |           | Fused in Pipeline 0 |
| +Expand(All)      | (p)-[i:INVENTED_BY]->(sc)                                   |               0 |     0 |        0 |               |                        |           | Fused in Pipeline 0 |
| +Filter           | p.born = $autoint_2 AND p:Pioneer                            |               0 |     0 |        2 |               |                        |           | Fused in Pipeline 0 |
| +Expand(All)      | (c)<-[anon_1:LIVES_IN]-(p)                                  |               1 |     1 |        3 |               |                        |           | Fused in Pipeline 0 |
| +Filter           | c:City                                                       |               1 |     1 |        1 |               |                        |           | Fused in Pipeline 0 |
| +Expand(All)      | (cc)<-[anon_2:PART_OF]-(c)                                  |               1 |     1 |        2 |               |                        |           | Fused in Pipeline 0 |
| +NodeIndexSeek    | BTREE INDEX cc:Country(formed) WHERE formed = $autoint_3    |               1 |     1 |        2 |       112      |                        |     3.981 | Fused in Pipeline 0 |
|                   |                                                              |               1 |     0 |        0 |               |                        |           |                     |

Total database accesses: 10, total allocated memory: 752

### Query using multiple index hints with a disjunction

Supplying multiple index hints can also be useful if the query contains a disjunction (OR) in the WHERE clause. This makes sure that all hinted indexes are used and the results are joined together with a Union and a Distinct afterwards.

---

619
Query

MATCH (country:Country)
USING INDEX country:Country(name)
USING INDEX country:Country(formed)
WHERE country.formed = 500 OR country.name STARTS WITH "A"
RETURN *

Query plan

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>country</td>
</tr>
<tr>
<td></td>
<td>+NodeIndexSeekByRange</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total database accesses: 3, total allocated memory: 1208

Cypher will usually provide a plan that uses all indexes for a disjunction without hints. It may, however, decide to plan a NodeByLabelScan instead, if the predicates appear to be not very selective. In this case, the index hints can be useful.

28.6.3. Scan hints

If your query matches large parts of an index, it might be faster to scan the label or relationship type and filter out rows that do not match. To do this, you can use USING SCAN variable:Label after the applicable MATCH clause for node indexes, and USING SCAN variable:RELATIONSHIP_TYPE for relationship indexes. This will force Cypher to not use an index that could have been used, and instead do a label scan/relationship type scan. You can use the same hint to enforce a starting point where no index is applicable.
Hinting a label scan

Query

```
MATCH (s:Scientist {born: 1850})-[[:RESEARCHED]->
(sc:Science)[:-[:INVENTED_BY (year: 560)]-]
(p:Pioneer {born: 525})-[[:LIVES_IN]->
(c:City)-[:PART_OF]->
(cc:Country {formed: 411})
USING SCAN s:Scientist
RETURN *
```
## Query plan

Compiler CYPHER 4.4
Planner COST
Runtime PIPELINED
Runtime version 4.4

| Operator         | Details                                                   | Estimated Rows | Rows |
|------------------|-----------------------------------------------------------|----------------+------|
| DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Other               |
| +------------------+-----------------------------------------------------------+----------------+------|
| +ProduceResults  | c, cc, i, p, s, sc                                        |              0 |    0 |
| 0 |                |                        |           | Fused in Pipeline 0 |
| |                +-----------------------------------------------------------+----------------+------|
| +Filter          | cc.formed = $autoint_3 AND cc:Country                     |              0 |    0 |
| 0 |                |                        |           | Fused in Pipeline 0 |
| |                +-----------------------------------------------------------+----------------+------|
| +Expand(All)     | (c)-[anon_2:PART_OF]->(cc)                                |              0 |    0 |
| 0 |                |                        |           | Fused in Pipeline 0 |
| |                +-----------------------------------------------------------+----------------+------|
| +Filter          | c:City                                                    |              0 |    0 |
| 0 |                |                        |           | Fused in Pipeline 0 |
| |                +-----------------------------------------------------------+----------------+------|
| +Expand(All)     | (p)-[anon_1:LIVES_IN]->(c)                                |              0 |    0 |
| 0 |                |                        |           | Fused in Pipeline 0 |
| |                +-----------------------------------------------------------+----------------+------|
| +Filter          | i.year = $autoint_1 AND p.born = $autoint_2 AND p:Pioneer |              0 |    0 |
| 1 |                |                        |           | Fused in Pipeline 0 |
| |                +-----------------------------------------------------------+----------------+------|
| +Expand(All)     | (sc)<-[i:INVENTED_BY]-(p)                                 |              1 |    1 |
| 3 |                |                        |           | Fused in Pipeline 0 |
| |                +-----------------------------------------------------------+----------------+------|
| +Filter          | sc:Science                                                |              1 |    1 |
| 200 |                |                        |           | Fused in Pipeline 0 |
| |                +-----------------------------------------------------------+----------------+------|
| +NodeByLabelScan | s:Scientist                                               |            100 |  100 |
| 101 |                | 112 |                   11/0 | 0.776 | Fused in Pipeline 0 |
| |                +-----------------------------------------------------------+----------------+------|

Total database accesses: 308, total allocated memory: 208

Hinting a relationship type scan
MATCH (s:Scientist {born: 1850})-[:RESEARCHED]->
(sc:Science)<-[i:INVENTED_BY {year: 560}]-
(p:Pioneer {born: 525})-[:LIVES_IN]->
(c:City)-[:PART_OF]->
(cc:Country {formed: 411})
USING SCAN i:INVENTED_BY
RETURN *
Query plan

Compiler CYPHER 4.4
Planner COST
Runtime PIPELINED

Runtime version 4.4

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimated Rows</td>
<td>Rows</td>
</tr>
<tr>
<td>+ProduceResults</td>
<td>c, cc, i, p, s, sc</td>
</tr>
<tr>
<td>+Filter</td>
<td>cc.formed = $autoint_3 AND cc:Country</td>
</tr>
<tr>
<td>+Expand(All)</td>
<td>(c)-[anon_2:PART_OF]-&gt;(cc)</td>
</tr>
<tr>
<td>+Filter</td>
<td>c:City</td>
</tr>
<tr>
<td>+Expand(All)</td>
<td>(p)-[anon_1:LIVES_IN]-&gt;(c)</td>
</tr>
<tr>
<td>+Filter</td>
<td>s.born = $autoint_0 AND s:Scientist</td>
</tr>
<tr>
<td>+Expand(All)</td>
<td>(sc)&lt;-[anon_0:RESEARCHED]-(s)</td>
</tr>
<tr>
<td>+Filter</td>
<td>i.year = $autoint_1 AND p.born = $autoint_2 AND sc:Science AND p:Pioneer</td>
</tr>
<tr>
<td>+DirectedRelationshipTypeScan</td>
<td>(p)-[i:INVENTED_BY]-&gt;(sc)</td>
</tr>
</tbody>
</table>

Total database accesses: 405, total allocated memory: 200
Query using multiple scan hints with a disjunction

Supplying multiple scan hints can also be useful if the query contains a disjunction (OR) in the WHERE clause. This makes sure that all involved label predicates are solved by a NodeByLabelScan and the results are joined together with a Union and a Distinct afterwards.

**Query**

```
MATCH (person)
USING SCAN person:Pioneer
USING SCAN person:Scientist
WHERE person:Pioneer OR person:Scientist
RETURN *
```

**Query plan**

```
Compiler CYPHER 4.4
Planner COST
Runtime PIPELINED
Runtime version 4.4
+------------------------+-----------+------------+---------------+-------+----------------+
| Operator             | Details          | Estimated Rows | Rows | DB Hits | Memory (Bytes) |
|-----------------------|------------------+----------------+------+---------+----------------+
| +ProduceResults       | person           |            180 |  200 |       0 |                |
| 4/0 | 2.622 | person ASC | In Pipeline 2 |
| +OrderedDistinct      | person           |            180 |  200 |       0 |             32 |
| 0/0 | 0.734 | person ASC | In Pipeline 2 |
| +OrderedUnion         |                  |            200 |  200 |       0 |           1128 |
| 0/0 | 1.264 | person ASC | In Pipeline 2 |
| +NodeByLabelScan      | person:Scientist |            100 |  100 | 101 |            112 |
| 1/0 | 0.294 | person ASC | In Pipeline 1 |
| +NodeByLabelScan      | person:Pioneer   |            100 |  100 | 101 |            112 |
| 1/0 | 0.729 | person ASC | In Pipeline 0 |
| Total database accesses: 202, total allocated memory: 1320 |
```

Cypher will usually provide a plan that uses scans for a disjunction without hints. It may, however, decide to plan an AllNodeScan followed by a Filter instead, if the label predicates appear to be not very selective. In this case, the scan hints can be useful.

### 28.6.4. 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, but not on the p node. By supplying a join hint in addition to the index hints, we can enforce the join to happen on the p node.

Query

```sql
MATCH (s:Scientist {born: 1850})-[[:RESEARCHED]]->
(sc:Science)<-[[:INVENTED_BY {year: 560}]]->
(p:Pioneer {born: 525})-[[:LIVES_IN]]->
(c:City)-[:PART_OF]->
(cc:Country {formed: 411})
USING INDEX s:Scientist(born)
USING INDEX cc:Country(former)
USING JOIN ON p
RETURN *
```
Hinting a join for an OPTIONAL MATCH

A join hint can also be used to force the planner to pick a `NodeLeftOuterHashJoin` or `NodeRightOuterHashJoin` to solve an OPTIONAL MATCH. In most cases, the planner will rather use an `OptionalExpand`.
Query

```cypher
MATCH (s:Scientist {born: 1850})
OPTIONAL MATCH (s)-[:RESEARCHED]-(sc:Science)
RETURN *
```

Without any hint, the planner did not use a join to solve the `OPTIONAL MATCH`.

Query plan

```
Compiler CYPHER 4.4
Planner COST
Runtime PIPELINED
Runtime version 4.4
```

| Operator                  | Details                                               | Estimated Rows | Rows |
|---------------------------+-------------------------------------------------------+----------------+------|
| +ProduceResults           | s, sc                                                 |              1 |    1 |
|                           |                                                      |                |      |
| +OptionalExpand(All)      | (s)-[anon_0:RESEARCHED]->(sc) WHERE sc:Science        |              1 |    1 |
|                           |                                                      |                |      |
| +NodeIndexSeek            | BTREE INDEX s:Scientist(born) WHERE born = $autoint_0 |              1 |    1 |
|                           |                                                      |                |      |

Total database accesses: 5, total allocated memory: 176

Query

```cypher
MATCH (s:Scientist {born: 1850})
OPTIONAL MATCH (s)-[:RESEARCHED]-(sc:Science)
USING JOIN ON s
RETURN *
```

Now the planner uses a join to solve the `OPTIONAL MATCH`. 
## Query plan

Compiler CYPHER 4.4  
Planner COST  
Runtime PIPELINED  
Runtime version 4.4

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
<th>Estimated Rows</th>
<th>Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB Hits</td>
<td>Memory (Bytes)</td>
<td>Page Cache Hits/Misses</td>
<td>Time (ms)</td>
</tr>
<tr>
<td>+ProduceResults</td>
<td>s, sc</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>2/0</td>
<td>0.123</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+NodeLeftOuterHashJoin</td>
<td>s</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>4864</td>
<td>4/0</td>
<td>4.057</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Expand(All)</td>
<td>(sc)&lt;-[anon_0:RESEARCHED]-(s)</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>300</td>
<td></td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+NodeByLabelScan</td>
<td>sc:Science</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>101</td>
<td></td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+NodeIndexSeek</td>
<td>BTREE INDEX s:Scientist(born) WHERE born = $autoint_0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>1/0</td>
<td>0.276</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Total database accesses: 403, total allocated memory: 4944

### 28.6.5. PERIODIC COMMIT query hint

The **PERIODIC COMMIT** query hint will be removed in the next major release. It is recommended to use **CALL { ... } IN TRANSACTIONS** instead.

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.
The **USE clause** can not be used together with the **PERIODIC COMMIT** query hint.

Queries with the **PERIODIC COMMIT** query hint can not be routed by **Server-side routing**. Such queries must rely on standard client-side routing, done by the Neo4j Driver.
Chapter 29. Execution plans

This section describes the characteristics of query execution plans and provides details about each of the operators.

Introduction

The task of executing a query is decomposed into operators, each of which implements a specific piece of work. The operators are combined into a tree-like structure called an execution plan. Each operator in the execution plan is represented as a node in the tree. Each operator takes as input zero or more rows, and produces as output zero or more rows. This means that the output from one operator becomes the input for the next operator. Operators that join two branches in the tree combine input from two incoming streams and produce a single output.

Evaluation model

Evaluation of the execution plan begins at the leaf nodes of the tree. Leaf nodes have no input rows and generally comprise operators such as scans and seeks. These operators obtain the data directly from the storage engine, thus incurring database hits. Any rows produced by leaf nodes are then piped into their parent nodes, which in turn pipe their output rows to their parent nodes and so on, all the way up to the root node. The root node produces the final results of the query.

Eager and lazy evaluation

In general, query evaluation is lazy: most operators pipe their output rows to their parent operators as soon as they are produced. This means that a child operator may not be fully exhausted before the parent operator starts consuming the input rows produced by the child.

However, some operators, such as those used for aggregation and sorting, need to aggregate all their rows before they can produce output. Such operators need to complete execution in its entirety before any rows are sent to their parents as input. These operators are called eager operators, and are denoted as such in Execution plan operators at a glance. Eagerness can cause high memory usage and may therefore be the cause of query performance issues.

Statistics

Each operator is annotated with statistics.

Rows

The number of rows that the operator produced. This is only available if the query was profiled.

EstimatedRows

This is the estimated number of rows that is expected to be produced by the operator. The estimate is an approximate number based on the available statistical information. 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 in Database hits (DbHits).

Page Cache Hits, Page Cache Misses, Page Cache Hit Ratio

These metrics are only shown for some queries when using Neo4j Enterprise Edition. The page cache is used to cache data and avoid accessing disk, so having a high number of hits and a low number of misses will typically make the query run faster. Whenever several operators are fused together for more efficient execution we can no longer associate this metric with a given operator and then nothing will appear here.

Time

Time is only shown for some operators when using the pipelined runtime. The number shown is the time in milliseconds it took to execute the given operator. Whenever several operators are fused together for more efficient execution we can no longer associate a duration with a given operator and then nothing will appear here.

To produce an efficient plan for a query, the Cypher query planner requires information about the Neo4j database. This information includes which indexes and constraints are available, as well as various statistics maintained by the database. The Cypher query planner 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. The number of relationships by type.
3. Selectivity per index.
4. The number of relationships by type, ending with or starting from a node with a specific label.

Information about how the statistics are kept up to date, as well as configuration options for managing query replanning and caching, can be found in the Operations Manual → Statistics and execution plans.

Query tuning describes how to tune Cypher queries. In particular, see Query tuning for how to view the execution plan for a query and Planner hints and the USING keyword for how to use hints to influence the decisions of the planner when building an execution plan for a query.

For a deeper understanding of how each operator works, refer to Execution plan operators at a glance and the linked sections per operator. Please remember that the statistics of the particular database where the queries run will decide the plan used. There is no guarantee that a specific query will always be solved with the same plan.

29.1. Execution plan operators at a glance

This section contains the 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.
- **Eager operators** accumulate all their rows before piping them to the next operator.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Leaf?</th>
<th>Updating?</th>
<th>Considerations</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllNodesScan</td>
<td>Reads all nodes from the node store.</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Anti</td>
<td>Tests for the absence of a pattern.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AntiSemiApply</td>
<td>Performs a nested loop. Tests for the absence of a pattern predicate.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apply</td>
<td>Performs a nested loop. Yields rows from both the left-hand and right-hand side operators.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Argument</td>
<td>Indicates the variable to be used as an argument to the right-hand side of an <strong>Apply</strong> operator.</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AssertSameNode</td>
<td>Used to ensure that no unique constraints are violated.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AssertingMultiNodeIndexSeek</td>
<td>Used to ensure that no unique constraints are violated.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CacheProperties</td>
<td>Reads node or relationship properties and caches them.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CartesianProduct</td>
<td>Produces a cartesian product of the inputs from the left-hand and right-hand operators.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Create</td>
<td>Creates nodes and relationships.</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CreateIndex</td>
<td>Creates an index for either nodes or relationships.</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Leaf?</td>
<td>Updating?</td>
<td>Considerations</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------</td>
<td>-----------</td>
<td>-----------------</td>
</tr>
<tr>
<td>CreateNodeKeyConstraint</td>
<td>Creates a node key constraint on a set of properties for all nodes with a certain label.</td>
<td></td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>CreateNodePropertyExistenceConstraint</td>
<td>Creates an existence constraint on a property for all nodes with a certain label.</td>
<td></td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>CreateRelationshipPropertyExistenceConstraint</td>
<td>Creates an existence constraint on a property for all relationships of a certain type.</td>
<td></td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>CreateUniqueConstraint</td>
<td>Creates a unique constraint on a set of properties for all nodes with a certain label.</td>
<td></td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Delete</td>
<td>Deletes a node or relationship.</td>
<td></td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>DetachDelete</td>
<td>Deletes a node and its relationships.</td>
<td></td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>DirectedRelationshipByIdSeek</td>
<td>Reads one or more relationships by id from the relationship store.</td>
<td></td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>DirectedRelationshipIndexContainsScan</td>
<td>Examines all values stored in an index, searching for entries containing a specific string; for example, in queries including CONTAINS.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DirectedRelationshipIndexEndsWithScan</td>
<td>Examines all values stored in an index, searching for entries ending in a specific string; for example, in queries containing ENDS WITH.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DirectedRelationshipIndexScan</td>
<td>Examines all values stored in an index, returning all relationships and their start and end nodes with a particular relationship type and a specified property.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Leaf?</td>
<td>Updating?</td>
<td>Considerations</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>-------</td>
<td>-----------</td>
<td>-----------------</td>
</tr>
<tr>
<td>DirectedRelationshipIndexSeek</td>
<td>Finds relationships and their start and end nodes using an index seek.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DirectedRelationshipIndexSeekByRange</td>
<td>Finds relationships and their start and end nodes using an index seek where the value of the property matches a given prefix string.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DirectedRelationshipTypeScan</td>
<td>Fetches all relationships and their start and end nodes with a specific type from the relationship type index.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distinct</td>
<td>Drops duplicate rows from the incoming stream of rows.</td>
<td></td>
<td></td>
<td>Eager</td>
</tr>
<tr>
<td>DoNothingIfExists(CONSTRAINT)</td>
<td>Checks if a constraint already exists, if it does then it stops the execution, if not it continues.</td>
<td></td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>DoNothingIfExists(INDEX)</td>
<td>Checks if an index already exists, if it does then it stops the execution, if not it continues.</td>
<td></td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>DropConstraint</td>
<td>Drops a constraint using its name.</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>DropIndex</td>
<td>Drops an index from a property for all nodes with a certain label.</td>
<td>Yes</td>
<td>Yes</td>
<td>Deprecated</td>
</tr>
<tr>
<td>DropIndex</td>
<td>Drops an index using its name.</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>DropNodeKeyConstraint</td>
<td>Drops a node key constraint from a set of properties for all nodes with a certain label.</td>
<td>Yes</td>
<td>Yes</td>
<td>Deprecated</td>
</tr>
<tr>
<td>DropNodePropertyExistenceConstraint</td>
<td>Drops an existence constraint from a property for all nodes with a certain label.</td>
<td>Yes</td>
<td>Yes</td>
<td>Deprecated</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Leaf?</td>
<td>Updating?</td>
<td>Considerations</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>-------</td>
<td>-----------</td>
<td>----------------</td>
</tr>
<tr>
<td>DropRelationshipPropertyExistenceConstraint</td>
<td>Drops an existence constraint from a property for all relationships of a certain type.</td>
<td>Yes</td>
<td>Yes</td>
<td>Deprecated</td>
</tr>
<tr>
<td>DropUniqueConstraint</td>
<td>Drops a unique constraint from a set of properties for all nodes with a certain label.</td>
<td>Yes</td>
<td>Yes</td>
<td>Deprecated</td>
</tr>
<tr>
<td>Eager</td>
<td>For isolation purposes, Eager ensures that operations affecting subsequent operations are executed fully for the whole dataset before continuing execution.</td>
<td></td>
<td></td>
<td>Eager</td>
</tr>
<tr>
<td>EagerAggregation</td>
<td>Evaluates a grouping expression.</td>
<td></td>
<td></td>
<td>Eager</td>
</tr>
<tr>
<td>EmptyResult</td>
<td>Eagerly loads all incoming data and discards it.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EmptyRow</td>
<td>Returns a single row with no columns.</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ExhaustiveLimit</td>
<td>The ExhaustiveLimit operator is similar to the Limit operator, but always exhausts the input. Used when combining LIMIT and updates.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expand(All)</td>
<td>Traverses incoming or outgoing relationships from a given node.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expand(Into)</td>
<td>Finds all relationships between two nodes.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Filter</td>
<td>Filters each row coming from the child operator, only passing through rows that evaluate the predicates to true.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Leaf?</td>
<td>Updating?</td>
<td>Considerations</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------</td>
<td>-----------</td>
<td>----------------</td>
</tr>
<tr>
<td>Foreach</td>
<td>Performs a nested loop. Yields rows from the left-hand operator and discards rows from the right-hand operator.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LetAntiSemiApply</td>
<td>Performs a nested loop. Tests for the absence of a pattern predicate in queries containing multiple pattern predicates.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LetSelectOrAntiSemiApply</td>
<td>Performs a nested loop. Tests for the absence of a pattern predicate that is combined with other predicates.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LetSelectOrSemiApply</td>
<td>Performs a nested loop. Tests for the presence of a pattern predicate that is combined with other predicates.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LetSemiApply</td>
<td>Performs a nested loop. Tests for the presence of a pattern predicate in queries containing multiple pattern predicates.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Limit</td>
<td>Returns the first 'n' rows from the incoming input.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LoadCSV</td>
<td>Loads data from a CSV source into the query.</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LockingMerge</td>
<td>Similar to the Merge operator but will lock the start and end node when creating a relationship if necessary.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Merge</td>
<td>The Merge operator will either read or create nodes and/or relationships.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MultiNodeIndexSeek</td>
<td>Finds nodes using multiple index seeks.</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Leaf?</td>
<td>Updating?</td>
<td>Considerations</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------</td>
<td>-----------</td>
<td>-----------------</td>
</tr>
<tr>
<td>NodeByIdSeek</td>
<td>Reads one or more nodes by ID from the node store.</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NodeByLabelScan</td>
<td>Fetches all nodes with a specific label from the node label index.</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NodeCountFromCountStore</td>
<td>Uses the count store to answer questions about node counts.</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NodeHashJoin</td>
<td>Executes a hash join on node ID.</td>
<td></td>
<td>Eager</td>
<td></td>
</tr>
<tr>
<td>NodeIndexContainsScan</td>
<td>Examines all values stored in an index, searching for entries containing a specific string.</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NodeIndexEndsWithScan</td>
<td>Examines all values stored in an index, searching for entries ending in a specific string.</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NodeIndexScan</td>
<td>Examines all values stored in an index, returning all nodes with a particular label with a specified property.</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NodeIndexSeek</td>
<td>Finds nodes using an index seek.</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NodeIndexSeekByRange</td>
<td>Finds nodes using an index seek where the value of the property matches the given prefix string.</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NodeLeftOuterHashJoin</td>
<td>Executes a left outer hash join.</td>
<td></td>
<td>Eager</td>
<td></td>
</tr>
<tr>
<td>NodeRightOuterHashJoin</td>
<td>Executes a right outer hash join.</td>
<td></td>
<td>Eager</td>
<td></td>
</tr>
<tr>
<td>NodeUniqueIndexSeek</td>
<td>Finds nodes using an index seek within a unique index.</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NodeUniqueIndexSeekByRange</td>
<td>Finds nodes using an index seek within a unique index where the value of the property matches the given prefix string.</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Leaf?</td>
<td>Updating?</td>
<td>Considerations</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------</td>
<td>-----------</td>
<td>-----------------</td>
</tr>
<tr>
<td>Optional</td>
<td>Yields a single row with all columns set to null if no data is returned by its source.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OptionalExpand(All)</td>
<td>Traverses relationships from a given node, producing a single row with the relationship and end node set to null if the predicates are not fulfilled.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OptionalExpand(Into)</td>
<td>Traverses all relationships between two nodes, producing a single row with the relationship and end node set to null if no matching relationships are found (the start node is the node with the smallest degree).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OrderedAggregation</td>
<td>Like EagerAggregation but relies on the ordering of incoming rows. Is not eager.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OrderedDistinct</td>
<td>Like Distinct but relies on the ordering of incoming rows.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PartialSort</td>
<td>Sorts a row by multiple columns if there is already an ordering.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PartialTop</td>
<td>Returns the first 'n' rows sorted by multiple columns if there is already an ordering.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ProcedureCall</td>
<td>Calls a procedure.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ProduceResults</td>
<td>Prepares the result so that it is consumable by the user.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ProjectEndpoints</td>
<td>Projects the start and end node of a relationship.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Leaf?</td>
<td>Updating?</td>
<td>Considerations</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------</td>
<td>-----------</td>
<td>----------------</td>
</tr>
<tr>
<td>Projection</td>
<td>Evaluates a set of expressions, producing a row with the results thereof.</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RelationshipCountFromCountStore</td>
<td>Uses the count store to answer questions about relationship counts.</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RemoveLabels</td>
<td>Deletes labels from a node.</td>
<td></td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>RollUpApply</td>
<td>Performs a nested loop. Executes a pattern expression or pattern comprehension.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SelectOrAntiSemiApply</td>
<td>Performs a nested loop. Tests for the absence of a pattern predicate if an expression predicate evaluates to <code>false</code>.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SelectOrSemiApply</td>
<td>Performs a nested loop. Tests for the presence of a pattern predicate if an expression predicate evaluates to <code>false</code>.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SemiApply</td>
<td>Performs a nested loop. Tests for the presence of a pattern predicate.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SetLabels</td>
<td>Sets labels on a node.</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SetNodePropertiesFromMap</td>
<td>Sets properties from a map on a node.</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SetProperty</td>
<td>Sets a property on a node or relationship.</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SetRelationshipPropertiesFromMap</td>
<td>Sets properties from a map on a relationship.</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ShortestPath</td>
<td>Finds one or all shortest paths between two previously matches node variables.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Leaf?</td>
<td>Updating?</td>
<td>Considerations</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------</td>
<td>-----------</td>
<td>----------------</td>
</tr>
<tr>
<td>ShowConstraints</td>
<td>Lists the available constraints.</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ShowFunctions</td>
<td>Lists the available functions.</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ShowIndexes</td>
<td>Lists the available indexes.</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ShowProcedures</td>
<td>Lists the available procedures.</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ShowTransactions</td>
<td>Lists the available transactions on the current server.</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Skip</td>
<td>Skips 'n' rows from the incoming rows.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sort</td>
<td>Sorts rows by a provided key.</td>
<td></td>
<td>Eager</td>
<td></td>
</tr>
<tr>
<td>TerminateTransactions</td>
<td>Terminate transactions with the given IDs.</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Top</td>
<td>Returns the first 'n' rows sorted by a provided key.</td>
<td></td>
<td>Eager</td>
<td></td>
</tr>
<tr>
<td>TriadicBuild</td>
<td>The TriadicBuild operator is used in conjunction with TriadicFilter to solve triangular queries.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TriadicFilter</td>
<td>The TriadicFilter operator is used in conjunction with TriadicBuild to solve triangular queries.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TriadicSelection</td>
<td>Solves triangular queries, such as the very common 'find my friend-of-friends that are not already my friend'.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UndirectedRelationshipByIdSeek</td>
<td>Reads one or more relationships by ID from the relationship store.</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Leaf?</td>
<td>Updating?</td>
<td>Considerations</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------</td>
<td>-----------</td>
<td>-----------------</td>
</tr>
<tr>
<td>UndirectedRelationshipIndexContainsScan</td>
<td>Examines all values stored in an index, searching for entries containing a specific string; for example, in queries including <code>CONTAINS</code>.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UndirectedRelationshipIndexEndsWithScan</td>
<td>Examines all values stored in an index, searching for entries ending in a specific string; for example, in queries containing <code>ENDS WITH</code>.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UndirectedRelationshipIndexScan</td>
<td>Examines all values stored in an index, returning all relationships and their start and end nodes with a particular relationship type and a specified property.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UndirectedRelationshipIndexSeek</td>
<td>Finds relationships and their start and end nodes using an index seek.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UndirectedRelationshipIndexSeekByRange</td>
<td>Finds relationships and their start and end nodes using an index seek where the value of the property matches a given prefix string.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UndirectedRelationshipTypeScan</td>
<td>Fetches all relationships and their start and end nodes with a specific type from the relationship type index.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Union</td>
<td>Concatenates the results from the right-hand operator with the results from the left-hand operator.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unwind</td>
<td>Returns one row per item in a list.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ValueHashJoin</td>
<td>Executes a hash join on arbitrary values.</td>
<td></td>
<td></td>
<td><code>Eager</code></td>
</tr>
<tr>
<td>VarLengthExpand(All)</td>
<td>Traverses variable-length relationships from a given node.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 29.2. Database hits (DbHits)

This section contains an overview of actions that triggers 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 one or more database hits:

- **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**

---

The presented value can vary slightly depending on the Cypher runtime that was used to execute the query. In the pipelined runtime the number of database hits will typically be higher since it uses a more accurate way of measuring.

### 29.3. Execution plan operators in detail

All operators are listed here, grouped by the similarity of their characteristics.

Certain operators are only used by a subset of the runtimes that Cypher can choose from. If that is the case, the example queries will be prefixed with an option to choose one of these runtimes.

#### 29.3.1. 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.
29.3.2. Directed Relationship Index Scan

The DirectedRelationshipIndexScan operator examines all values stored in an index, returning all relationships and their start and end nodes with a particular relationship type and a specified property.

Query

MATCH ()-[r: WORKS_IN]->() WHERE r.title IS NOT NULL RETURN r
29.3.3. Undirected Relationship Index Scan

The UndirectedRelationshipIndexScan operator examines all values stored in an index, returning all relationships and their start and end nodes with a particular relationship type and a specified property.

**Query**

```
MATCH ()-[r:WORKS_IN]-() WHERE r.title IS NOT NULL RETURN r
```
29.3.4. Directed Relationship Index Seek

The DirectedRelationshipIndexSeek operator finds relationships and their start and end nodes using an index seek. The relationship variable and the index used are shown in the arguments of the operator.

Query

```
MATCH (candidate)-[r:WORKS_IN]->() WHERE r.title = 'chief architect' RETURN candidate
```
29.3.5. Undirected Relationship Index Seek

The UndirectedRelationshipIndexSeek operator finds relationships and their start and end nodes using an index seek. The relationship variable and the index used are shown in the arguments of the operator.

Query

```cypher
MATCH (candidate)-[r:WORKS_IN]-() WHERE r.title = 'chief architect' RETURN candidate
```
29.3.6. 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**

```cypher
MATCH (n1)-[r]->()
WHERE id(r) = 0
RETURN r, n1
```
29.3.7. 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 4.4
Planner COST
Runtime PIPELINED
Runtime version 4.4

| Operator                        | Details                                    | Estimated Rows | Rows | Other               |
|-------------------------------|--------------------------------------------|----------------+------|---------------------|
| ProduceResults                 | r, n1                                      | 2              | 2    | Fused in Pipeline 0 |
| +-------------------------------|--------------------------------------------|----------------+------|---------------------|
| UndirectedRelationshipByIdSeek | (n1)-[r]-(anon_0) WHERE id(r) = $autoint_0 | 2              | 2    | Fused in Pipeline 0 |

Total database accesses: 1, total allocated memory: 176
29.3.8. Directed Relationship Index Contains Scan

The DirectedRelationshipIndexContainsScan 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 type scan using DirectedRelationshipTypeScan, and a property store filter.

Query

MATCH ()-[r:WORKS_IN]->() WHERE r.title CONTAINS 'senior' RETURN r

Query Plan

Compiler CYPHER 4.4
Planner COST
Runtime PIPELINED
Runtime version 4.4

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimated Rows</td>
<td>Rows</td>
</tr>
<tr>
<td>+ProduceResults</td>
<td>r</td>
</tr>
<tr>
<td>+DirectedRelationshipIndexContainsScan</td>
<td>BTREE INDEX (anon_0)-[r:WORKS_IN(title)]-&gt;(anon_1) WHERE title CONTAINS $autostring_0</td>
</tr>
</tbody>
</table>

Total database accesses: 9, total allocated memory: 176

29.3.9. Undirected Relationship Index Contains Scan

The UndirectedRelationshipIndexContainsScan 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 type scan using DirectedRelationshipTypeScan, and a property store filter.

Query

MATCH ()-[r:WORKS_IN]-() WHERE r.title CONTAINS 'senior' RETURN r
29.3.10. Directed Relationship Index Ends With Scan

The `DirectedRelationshipIndexEndsWithScan` 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 ()-[r:WORKS_IN]->() WHERE r.title ENDS WITH 'developer' RETURN r
```
### 29.3.11. Undirected Relationship Index Ends With Scan

The **UndirectedRelationshipIndexEndsWithScan** 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 ()-[r:WORKS_IN]()->() WHERE r.title ENDS WITH 'developer' RETURN r
```
29.3.12. Directed Relationship Index Seek By Range

The **DirectedRelationshipIndexSeekByRange** operator finds relationships and their start and end nodes using an index seek where the value of the property matches a given prefix string. **DirectedRelationshipIndexSeekByRange** can be used for **STARTS WITH** and comparison operators such as <, >, <= and >=.

**Query**

```
MATCH (candidate: Person)-[r:WORKS_IN]->(location) WHERE r.duration > 100 RETURN candidate
```
29.3.13. Undirected Relationship Index Seek By Range

The `UndirectedRelationshipIndexSeekByRange` operator finds relationships and their start and end nodes using an index seek where the value of the property matches a given prefix string. `UndirectedRelationshipIndexSeekByRange` can be used for `STARTS WITH` and comparison operators such as `<`, `>`, `<=` and `>=`.

Query

```cypher
MATCH (candidate: Person)-[r:WORKS_IN]-(location) WHERE r.duration > 100 RETURN candidate
```
29.3.14. Directed Relationship Type Scan

The DirectedRelationshipTypeScan operator fetches all relationships and their start and end nodes with a specific type from the relationship type index.

Query

```
MATCH ()-[r: FRIENDS_WITH]->() RETURN r
```
29.3.15. Undirected Relationship Type Scan

The UndirectedRelationshipTypeScan operator fetches all relationships and their start and end nodes with a specific type from the relationship type index.

Query

\[
\text{MATCH} \ (\) \-[r: \text{FRIENDS\_WITH}]-() \ \text{RETURN} \ r
\]
29.3.16. 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 4.4  
Planner COST  
Runtime PIPELINED  
Runtime version 4.4

```
<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Memory (Bytes)</th>
<th>Page Cache Hits/Misses</th>
<th>Time (ms)</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>n</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td>0.340</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fused in Pipeline 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+NodeByIdSeek</td>
<td>n WHERE id(n) = $autoint_0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>112</td>
<td></td>
<td>0.340</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fused in Pipeline 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Total database accesses: 1, total allocated memory: 176

29.3.17. 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
```

658
### 29.3.18. Node Index Seek

The **NodeIndexSeek** operator finds nodes using an index seek. The node variable and the index used are 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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Memory (Bytes)</th>
<th>Page Cache Hits/Misses</th>
<th>Time (ms)</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>location</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>Fused in Pipeline 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+NodeIndexSeek</td>
<td>BTREE INDEX location:Location(name) WHERE name = $autostring_0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td>1.090</td>
<td>Fused in Pipeline 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total database accesses: 2, total allocated memory: 176
29.3.19. Node Unique Index Seek

The NodeUniqueIndexSeek operator finds nodes using an index seek within a unique index. The node variable and the index used are 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

```cypher
MATCH (t:Team {name: 'Malmo'}) RETURN t
```

Query Plan

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory (Bytes)</td>
<td>Page Cache Hits/Misses</td>
<td>Time (ms)</td>
<td>Other</td>
<td></td>
</tr>
<tr>
<td>+ProduceResults</td>
<td>t</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fused in Pipeline 0</td>
</tr>
<tr>
<td>+NodeUniqueIndexSeek</td>
<td>UNIQUE t:Team(name) WHERE name = $autostring_0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>112</td>
<td>0/1</td>
<td>0.752</td>
<td>Fused in Pipeline 0</td>
</tr>
<tr>
<td>+-------------------------------------------+------------------------------------------------+----------------+------+---------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| Total database accesses: 1, total allocated memory: 176

29.3.20. Multi Node Index Seek

The MultiNodeIndexSeek operator finds nodes using multiple index seeks. It supports using multiple distinct indexes for different nodes in the query. The node variables and the indexes used are shown in the arguments of the operator.

The operator yields a cartesian product of all index seeks. For example, if the operator does two seeks and the first seek finds the nodes a1, a2 and the second b1, b2, b3, the MultiNodeIndexSeek will yield the rows (a1, b1), (a1, b2), (a1, b3), (a2, b1), (a2, b2), (a2, b3).

Query

```cypher
MATCH (location:Location {name: 'Malmo'}), (person:Person {name: 'Bob'}) RETURN location, person
```
29.3.21. Asserting Multi Node Index Seek

The **AssertingMultiNodeIndexSeek** 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

```mergesql
MERGE (t:Team {name: 'Engineering', id: 42})
```
### 29.3.22. 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

```cypher
MATCH (l:Location) WHERE l.name STARTS WITH 'Lon' RETURN l
```
29.3.23. 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 CYPER 4.4  
Planner COST  
Runtime PIPELINED  

Runtime version 4.4  

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
<th>Estimated Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>t</td>
<td>2</td>
</tr>
<tr>
<td>+NodeUniqueIndexSeekByRange</td>
<td>UNIQUE t:Team(name) WHERE name STARTS WITH $autostring_0</td>
<td>2</td>
</tr>
</tbody>
</table>

Total database accesses: 1, total allocated memory: 176

### 29.3.24. 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**

```cypher
MATCH (l:Location) WHERE l.name CONTAINS 'al' RETURN l
```
### Query Plan

- Compiler: CYPER 4.4
- Planner: COST
- Runtime: PIPELINED
- Runtime version: 4.4

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
<th>Estimated Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>l</td>
<td>0</td>
</tr>
<tr>
<td>+NodeIndexContainsScan</td>
<td>BTREE INDEX l:Location(name) WHERE name CONTAINS $autostring_0</td>
<td>0</td>
</tr>
</tbody>
</table>

Total database accesses: 3, total allocated memory: 176

#### 29.3.25. 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**

```cypher
MATCH (l:Location) WHERE l.name ENDS WITH 'al' RETURN l
```
29.3.26. Node Index Scan

The **NodeIndexScan** operator examines all values stored in an index, returning all nodes with a particular label and a specified property.

**Query**

```
MATCH (l:Location) WHERE l.name IS NOT NULL RETURN l
```
29.3.27. 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 4.4
Planner COST
Runtime PIPELINED
Runtime version 4.4

+------------------+-------------------------------------------------------+----------------+------+
| Operator          | Details                                               | Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Other               |
+------------------+-------------------------------------------------------+----------------+------+
| +ProduceResults   | p, q                                                  |              1 |    0 |        0 |                |                        |           | Fused in Pipeline 1 |
| |                +-------------------------------------------------------+----------------+------+
| +Apply            |                                                       |              1 |    0 |        0 |                |                        |           |                     |
| | |
| | +NodeIndexSeek   | BTREE INDEX q:Person(name) WHERE name = p.secondName  |              1 |    0 |        2 |            2152 |                    0/0 | 0.363 | Fused in Pipeline 1 |
| | |                |                                                        |              0 |    0 |        0 |                |                        |           |                     |
| | |                |                                                        |              0 |    0 |        0 |                |                        |           |                     |
| +NodeIndexSeek    | BTREE INDEX p:Person(name) WHERE name = $autostring_0 |              1 |    1 |        2 |            112 |                    0/1 | 0.850 | In Pipeline 0      |
| |                |                                                        |              0 |    0 |        0 |                |                        |           |                     |
| +------------------+-------------------------------------------------------+----------------+------+
Total database accesses: 2, total allocated memory: 2216
```

29.3.28. 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

CYpher runtime=sloected
MATCH (p:Person)
WHERE (p)-[:FRIENDS_WITH]->(:Person)
RETURN p.name

Query Plan

Compiler CYpher 4.4
Planner COST
Runtime SLOTTED
Runtime version 4.4

| Operator        | Details                             | Estimated Rows | Rows | DB Hits | Page Cache Hits/Misses |
|-----------------|-------------------------------------|----------------+------+---------+------------------------|
| +-----------------+-------------------------------------+----------------+------+---------+------------------------|
| +ProduceResults | p.name                               | 11             | 2    | 0       |                         |
| |                 +-------------------------------------+----------------+------+---------+------------------------|
| +Projection     | p.name AS 'p.name'                   | 11             | 2    | 2       |                         |
| |                 +-------------------------------------+----------------+------+---------+------------------------|
| +SemiApply      |                                     | 11             | 2    | 0       |                         |
| |                 +-------------------------------------+----------------+------+---------+------------------------|
| | +Filter       | anon_3:Person                        | 2              | 0    | 2       |                         |
| | |             +-------------------------------------+----------------+------+---------+------------------------|
| | +Expand(All)  | (p)-[anon_2:FRIENDS_WITH]->(anon_3) | 28             | 2    | 33      |                         |
| | |                 +-------------------------------------+----------------+------+---------+------------------------|
| | +Argument     | p                                     | 14             | 14   | 0       |                         |
| | |               +-------------------------------------+----------------+------+---------+------------------------|
| | +Filter       | p:Person                             | 14             | 14   | 35      |                         |
| | |                 +-------------------------------------+----------------+------+---------+------------------------|
| | +AllNodesScan | p                                     | 35             | 35   | 36      |                         |
| +-----------------+-------------------------------------+----------------+------+---------+------------------------|
| Total database accesses: 108, total allocated memory: 64

29.3.29. 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 4.4
Planner COST
Runtime SLOTTED
Runtime version 4.4

Total database accesses: 141, total allocated memory: 968

29.3.30. Anti

The Anti operator tests for the absence of a pattern. If there are incoming rows, the Anti operator will yield no rows. If there are no incoming rows, the Anti operator will yield a single row.
Query

**CYPHER runtime=pipelined**

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

Query Plan

Compiler CYPHER 4.4
Planner COST
Runtime PIPELINED

Runtime version 4.4

| Operator          | Details                                                | Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Other               |
|-------------------|--------------------------------------------------------|----------------+------|---------+----------------+------------------------+-----------|---------------------|
| +ProduceResults   | `other.name`                                           |              4 |   13 | 0       |                |                    0/0 |     0.261 | In Pipeline 4       |
| +Projection       | other.name AS `other.name`                             |              4 |   13 | 26      |                |                    2/0 |     0.127 | In Pipeline 4       |
| +Apply            |                                                        |              4 |   13 | 0       |                |                    0/0 |           |                     |
| +Anti             |                                                        |             14 |   13 | 0       | 1256           |                    0/0 |     0.154 | In Pipeline 4       |
| +Limit            | 1                                                      |             14 |   14 | 0       | 3184           |                    1/0 |     6.201 | Fused in Pipeline 3 |
| +CartesianProduct |                                                        |             14 |   14 | 0       | 3664           |                    0/0 |     0.212 | In Pipeline 2       |
| +Filter           | other:Person                                           |             14 |   14 | 0       | 112            |                    0/1 |     0.512 | In Pipeline 0       |

Total database accesses: 119, total allocated memory: 6736
29.3.31. 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

```cypher
MATCH (other:Person)
WHERE (other)-[:FRIENDS_WITH]->(:Person) OR (other)-[:WORKS_IN]->(:Location)
RETURN other.name
```
### 29.3.32. 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
```
29.3.33. 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 \texttt{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 4.4
Planner COST
Runtime PIPELINED
Runtime version 4.4

+--------------------+-----------------------------------------+----------------+------+---------+
| Operator           | Details                                 | Estimated Rows | Rows | DB Hits |
|--------------------|-----------------------------------------|----------------+------+---------+
| +ProduceResults    | `other.name`                            | 11 | 2 | 0 |
| +----------------+-----------------------------------------+----------------+------+---------+
| +ProduceResults    | `other.name`                            | 11 | 2 | 0 |
| +----------------+-----------------------------------------+----------------+------+---------+
| +SelectOrSemiApply | other.age > $autoint_0                  | 14 | 2 | 0 |
| +----------------+-----------------------------------------+----------------+------+---------+
| +Limit           | 1                                       | 14 | 2 | 0 |
| +----------------+-----------------------------------------+----------------+------+---------+
| +Filter          | anon_3:Person                           | 2 | 2 | 2 |
| +----------------+-----------------------------------------+----------------+------+---------+
| +Expand(All)     | (other)-[anon_2:FRIENDS_WITH]->(anon_3) | 2 | 2 | 32 |
| +----------------+-----------------------------------------+----------------+------+---------+
| +Argument        | other                                   | 2 | 2 | 2 |
| +----------------+-----------------------------------------+----------------+------+---------+
| +AllNodesScan    | other                                   | 35 | 35 | 36 |
| +----------------+-----------------------------------------+----------------+------+---------+

Total database accesses: 74, total allocated memory: 5040
```

29.3.34. Select Or Anti Semi Apply

The \texttt{SelectOrAntiSemiApply} operator is used to evaluate \texttt{OR} between a predicate and a negative pattern predicate (i.e. a pattern predicate preceded with \texttt{NOT}), and is a variation of the \texttt{Apply} operator. If the predicate returns \texttt{true}, the pattern predicate is not tested. If the predicate returns \texttt{false} or \texttt{null}, \texttt{SelectOrAntiSemiApply} will instead test the pattern predicate.
29.3.35. 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

```
CYPHER runtime=slotted
MATCH (other:Person)
WHERE (other)-[[:FRIENDS_WITH]->(:Person) OR (other)-[:WORKS_IN]->(:Location) OR other.age = 5
RETURN other.name
```

### Query Plan

Compiler CYPHER 4.4
Planner COST
Runtime SLOTTED
Runtime version 4.4

| Operator              | Details                                 | Estimated Rows | Rows | DB Hits | Page Cache Hits/Misses |
|-----------------------|-----------------------------------------|----------------+------+---------+------------------------|
| +ProduceResults       | `other.name`                            |             13 |   14 |       0 | 0/0                      |
| +Projection           | other.name AS `other.name`              |             13 |   14 |      14 | 1/0                      |
| +SelectOrSemiApply    | anon_9                                  |             14 |   14 |      14 | 0/0                      |
| +Filter               | anon_7:Location                         |             14 |    0 |      12 | 0/0                      |
| +Expand(All)          | (other)-[anon_6:WORKS_IN]->(anon_7)     |             14 |   12 |      26 | 24/0                     |
| +Argument             | other                                   |             14 |   14 |      35 | 1/0                      |
| AllNodesScan          | other                                   |             35 |   35 |      36 | 1/0                      |

Total database accesses: 172, total allocated memory: 64
29.3.36. 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

```cypher
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 4.4  
**Planner** COST  
**Runtime** SLOTTED  

**Runtime version 4.4**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Page Cache Hits/Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td><code>other.name</code></td>
<td>12</td>
<td>14</td>
<td>0</td>
<td>0/0</td>
</tr>
<tr>
<td>+Projection</td>
<td>other.name AS <code>other.name</code></td>
<td>12</td>
<td>14</td>
<td>14</td>
<td>0/0</td>
</tr>
<tr>
<td>+SelectOrSemiApply</td>
<td>anon_9</td>
<td>14</td>
<td>14</td>
<td>0</td>
<td>0/0</td>
</tr>
<tr>
<td></td>
<td>+Filter</td>
<td>anon_7:Location</td>
<td>14</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>+Expand(All)</td>
<td>(other)-[anon_6:WORKS_IN]-&gt;(anon_7)</td>
<td>14</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+Argument</td>
<td>other</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>+LetSelectOrAntiSemiApply</td>
<td>other.age = $autoint_0</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>+Filter</td>
<td>anon_5:Person</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>+Expand(All)</td>
<td>(other)-[anon_4:FRIENDS_WITH]-&gt;(anon_5)</td>
<td>2</td>
<td>2</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+Argument</td>
<td>other</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>+Filter</td>
<td>other:Person</td>
<td>14</td>
<td>14</td>
<td>35</td>
</tr>
<tr>
<td>+AllNodesScan</td>
<td>other</td>
<td>35</td>
<td>35</td>
<td>36</td>
<td>1/0</td>
</tr>
</tbody>
</table>

Total database accesses: 143, total allocated memory: 64

### 29.3.37. Merge

The **Merge** operator will either read or create nodes and/or relationships.

If matches are found it will execute the provided **ON MATCH** operations foreach incoming row. If no matches
are found instead nodes and relationships are created and all **ON CREATE** operations are run.

**Query**

```
MERGE (p:Person {name: 'Andy'})
ON MATCH SET p.existed = true
ON CREATE SET p.existed = false
```

**Query Plan**

Compiler CYPHER 4.4  
Planner COST  
Runtime PIPELINED  
Runtime version 4.4

| Operator        | Details                                                                 | Estimated | |
|-----------------|-------------------------------------------------------------------------|-----------|
| +ProduceResults |                                                                         |           | |
| +EmptyResult    |                                                                         |           | |
| +Merge          | CREATE (p:Person {name: $autostring_0}), ON MATCH SET p.existed = true,  |
| | | | | Fused in Pipeline 0 | |
| +NodeIndexSeek  | BTREE INDEX p:Person(name) WHERE name = $autostring_0                   |           | |

Total database accesses: 4, total allocated memory: 176

29.3.38. Locking Merge

The **LockingMerge** operator is just like a normal **Merge** but will lock the start and end node when creating a relationship if necessary.

**Query**

```
MATCH (s:Person {name: 'me'}) MERGE (s)-[:FRIENDS_WITH]->(s)
```
### Query Plan

**Compiler CYPHER 4.4**

**Planner COST**

**Runtime PIPELINED**

**Runtime version 4.4**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Memory (Bytes)</th>
<th>Page Cache Hits/Misses</th>
<th>Time (ms)</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>Fused in Pipeline</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+EmptyResult</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>Fused in Pipeline</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Apply</td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+LockingMerge</td>
<td>CREATE (s)-[anon_0:FRIENDS_WITH]-&gt;(s), LOCK(s)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>0.300</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Expand(Into)</td>
<td>(s)-[anon_0:FRIENDS_WITH]-&gt;(s)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Argument</td>
<td>s</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>2152</td>
<td>3/0</td>
<td>0.827</td>
<td>Fused in Pipeline</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+NodeIndexSeek</td>
<td>BTREE INDEX s:Person(name) WHERE name = $autostring_0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>0.300</td>
<td></td>
</tr>
</tbody>
</table>
| Total database accesses: 11, total allocated memory: 2232

---

**29.3.39. 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**

```cypher
MATCH (p:Person)
RETURN p.name, [(p)-[:WORKS_IN]->(location) | location.name] AS cities
```
### Query Plan

**Compiler** CYPHER 4.4  
**Planner** COST  
**Runtime** SLOTTED  

**Runtime version** 4.4

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Page Cache Hits/Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>'p.name', cities</td>
<td>14</td>
<td>14</td>
<td>0</td>
<td>0/0</td>
</tr>
<tr>
<td>+Projection</td>
<td>p.name AS 'p.name'</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>0/0</td>
</tr>
<tr>
<td>+RollUpApply</td>
<td>cities, anon_0</td>
<td>14</td>
<td>14</td>
<td>0</td>
<td>0/0</td>
</tr>
<tr>
<td></td>
<td>+Projection</td>
<td>location.name AS anon_0</td>
<td>6</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>+Expand(All)</td>
<td>(p)-[anon_2:WORKS_IN]-&gt;(location)</td>
<td>6</td>
<td>15</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>+Argument</td>
<td>p</td>
<td>14</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>+Filter</td>
<td>p:Person</td>
<td>14</td>
<td>14</td>
<td>35</td>
</tr>
<tr>
<td>+AllNodesScan</td>
<td>p</td>
<td>35</td>
<td>35</td>
<td>36</td>
<td>1/0</td>
</tr>
</tbody>
</table>

Total database accesses: 133, total allocated memory: 64

---

### 29.3.40. 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)
```
29.3.41. Expand All

Given a start node, and depending on the pattern relationship, the Expand(All) operator will traverse incoming or outgoing relationships.

Query

```cypher
MATCH (p:Person {name: 'me'})-[[:FRIENDS_WITH]-]->(fof) RETURN fof
```
29.3.42. 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 4.4  
Planner COST  
Runtime PIPELINED  
Runtime version 4.4

| Operator          | Details                                                                 | Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Other               |
|-------------------|-------------------------------------------------------------------------|----------------+------+---------+----------------+------------------------+-----------+---------------------|
| +ProduceResults   | fof                                                                     | 0              | 0    | 0       |                |                        |           | Fused in Pipeline 0 |
| +Filter           | not anon_0 = anon_1                                                     | 0              | 0    | 0       |                |                        |           | Fused in Pipeline 0 |
| +Expand(Into)     | (p)-[anon_0:FRIENDS_WITH]->(fof)                                       | 0              | 0    | 0       | 0              |                        |           | Fused in Pipeline 0 |
| +Expand(All)      | (p)<-[anon_1]-(fof)                                                    | 3              | 0    | 0       |                |                        |           | Fused in Pipeline 0 |
| +NodeIndexSeek    | BTREE INDEX p:Person(name) WHERE name = $autostring_0                   | 2              | 1    | 1       | 112            | 2/1                   | 1.103     | Fused in Pipeline 0 |

Total database accesses: 5, total allocated memory: 192

### 29.3.43. 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**

```cypher
MATCH (p:Person)
OPTIONAL MATCH (p)-[works_in:WORKS_IN]->(l) WHERE works_in.duration > 180
RETURN p, l
```
29.3.44. 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

```cypher
MATCH (p:Person)-[:WORKS_IN]->(l) OPTIONAL MATCH (l)-->p RETURN p
```
### Query Plan

**Compiler:** CYPHER 4.4  
**Planner:** COST  
**Runtime:** PIPELINED  
**Runtime version:** 4.4

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Memory (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ProduceResults</td>
<td>p</td>
<td>15</td>
<td>15</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fused in Pipeline 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+OptionalExpand(Into)</td>
<td>(l)-[anon_0]-&gt;(p)</td>
<td>15</td>
<td>15</td>
<td>105</td>
<td>3352</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fused in Pipeline 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Expand(All)</td>
<td>(p)-[works_in:WORKS_IN]-&gt;(l)</td>
<td>15</td>
<td>15</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fused in Pipeline 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Filter</td>
<td>p:Person</td>
<td>14</td>
<td>14</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fused in Pipeline 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+AllNodesScan</td>
<td>p</td>
<td>35</td>
<td>35</td>
<td>36</td>
<td>112</td>
</tr>
<tr>
<td></td>
<td>6/0</td>
<td>1.941</td>
<td>Fused in Pipeline 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total database accesses: 160, total allocated memory: 3432

---

#### 29.3.45. VarLength Expand All

Given a start node, the **VarLengthExpand(All)** operator will traverse variable-length relationships.

**Query**

```cypher
MATCH (p:Person)-[:FRIENDS_WITH *1..2]-(q:Person) RETURN p, q
```
29.3.46. 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]-(q:Person) RETURN p
```
### Query Plan

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Memory (Bytes)</th>
<th>Page Cache Hits/Misses</th>
<th>Time (ms)</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>p</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+VarLengthExpand(Into)</td>
<td>(p)-[anon_0:FRIENDS_WITH*..2]-(p)</td>
<td>0</td>
<td>0</td>
<td>47</td>
<td>128</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Filter</td>
<td>p:Person</td>
<td>14</td>
<td>14</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+AllNodesScan</td>
<td>p</td>
<td>35</td>
<td>35</td>
<td>36</td>
<td>112</td>
<td></td>
<td>11.021</td>
<td>Fused in Pipeline 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total database accesses: 83, total allocated memory: 192

### 29.3.47. 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
```
29.3.48. Assert Same Node

The **AssertSameNode** operator is used to ensure that no unique constraints are violated in the slotted and interpreted runtime. 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

```
CYPHER runtime=slotted MERGE (t:Team {name: 'Engineering', id: 42})
```
### 29.3.49. Empty Result

The **EmptyResult** operator eagerly loads all incoming data and discards it.

**Query**

```cypher
CREATE (:Person)
```
### 29.3.50. 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**

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

**Query Plan**

| Operator          | Details         | Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Other               |
|-------------------|-----------------|----------------+------|---------+----------------+-------------------------+-----------|---------|
| +ProduceResults   |                 | 35             | 35    | 0        |                |                        |          |                    |
| Fused in Pipeline 0 |                |                |       |          |                |                        |          |                    |
| +AllNodesScan     | n               | 35             | 35    | 36       | 112            |                         | 3/0       | 0.734               |
| Fused in Pipeline 0 |                |                |       |          |                |                        |          |                    |

Total database accesses: 36, total allocated memory: 176
29.3.51. 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 4.4
Planner COST
Runtime PIPELINED
Runtime version 4.4
+-----------------+---------+----------------+------+---------+----------------+------------------------+
| Operator        | Details | Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses |
| Time (ms) | Other         |
+-----------------+---------+----------------+------+---------+----------------+------------------------+
| +ProduceResults | line    |             10 |    4 |       0 |                |                    0/0 |
| 0.726 | In Pipeline 1 |
| +LoadCSV        | line    |             10 |    4 |       0 |            104 |                        |
| In Pipeline 1 | |
| Total database accesses: 0, total allocated memory: 176 |
```

29.3.52. 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**

29.3.53. 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 (bob:Person {name: 'Bob'})-[[:WORKS_IN]]->(loc)<-[[:WORKS_IN]]-(matt:Person {name: 'Mattis'})
RETURN loc.name
```

Query Plan

```
Compiler CYPHER 4.4
Planner COST
Runtime PIPELINED
Runtime version 4.4

| Operator         | Details                                                  | Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Other               |
|------------------|----------------------------------------------------------|----------------+------+---------+----------------+--------------------+-----------+---------------------|
| +ProduceResults  | `loc.name`                                               |              10 |    0 |         |                |                    |     0.000 | In Pipeline 2       |
| |                +----------------------------------------------------------|----------------+------+---------+----------------+--------------------+-----------+---------------------|
| +Projection      | loc.name AS `loc.name`                                   |              10 |    0 |         |                |                    |     0.000 | In Pipeline 2       |
| |                +----------------------------------------------------------|----------------+------+---------+----------------+--------------------+-----------+---------------------|
| +Filter          | not anon_0 = anon_1                                       |              10 |    0 |         |                |                    |     0.000 | In Pipeline 2       |
| |                +----------------------------------------------------------|----------------+------+---------+----------------+--------------------+-----------+---------------------|
| +NodeHashJoin    | loc                                                      |              10 |    0 |         |            3680 |                    |     0.054 | In Pipeline 2       |
| |
| | +Expand(All)   | (matt)-[anon_1:WORKS_IN]->(loc)                           |              19 |    0 |         |                |                    |           | Fused in Pipeline 1 |
| | |              +----------------------------------------------------------|----------------+------+---------+----------------+--------------------+-----------+---------------------|
| | +NodeIndexSeek | BTREE INDEX matt:Person(name) WHERE name = $autostring_1 |              1 |    0 |         |            112 |                    |     0.453 | Fused in Pipeline 1 |
| | |
| | +Expand(All)     | (bob)-[anon_0:WORKS_IN]->(loc)                            |              19 |    1 |         |                |                    |           | Fused in Pipeline 0 |
| | |                +----------------------------------------------------------|----------------+------+---------+----------------+--------------------+-----------+---------------------|
| | +NodeIndexSeek   | BTREE INDEX bob:Person(name) WHERE name = $autostring_0  |              1 |    1 |         |            112 |                    |     0.697 | Fused in Pipeline 0 |
| | |
| | +ProduceResults  | `loc.name`                                               |              10 |    0 |         |                |                    |     0.000 | In Pipeline 2       |
| |                +----------------------------------------------------------|----------------+------+---------+----------------+--------------------+-----------+---------------------|
|
Total database accesses: 6, total allocated memory: 3872
```

29.3.54. 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 4.4
Planner COST
Runtime PIPELINED
Runtime version 4.4

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Memory (Bytes)</th>
<th>Page Cache Hits/Misses</th>
<th>Time (ms)</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>p, q</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>0.000</td>
<td>In Pipeline 2</td>
</tr>
<tr>
<td>+ValueHashJoin</td>
<td>p.age = q.age</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>344</td>
<td></td>
<td></td>
<td>In Pipeline 2</td>
</tr>
<tr>
<td>+Filter</td>
<td>q:Person</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>Fused in Pipeline 1</td>
</tr>
<tr>
<td>+AllNodesScan</td>
<td>q</td>
<td>35</td>
<td>0</td>
<td>0</td>
<td>112</td>
<td></td>
<td>0.000</td>
<td>Fused in Pipeline 1</td>
</tr>
<tr>
<td>+Filter</td>
<td>p:Person</td>
<td>14</td>
<td>14</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>Fused in Pipeline 0</td>
</tr>
<tr>
<td>+AllNodesScan</td>
<td>p</td>
<td>35</td>
<td>35</td>
<td>36</td>
<td>112</td>
<td></td>
<td>1.270</td>
<td>Fused in Pipeline 0</td>
</tr>
</tbody>
</table>

Total database accesses: 36, total allocated memory: 648

#### 29.3.55. 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 \( (a) \rightarrow (b:Person) \). If \( (a:Person) \) returns fewer results than \( (a) \rightarrow (b:Person) \), a left outer join — indicated by NodeLeftOuterHashJoin — is planned. On the other hand, if \( (a:Person) \) returns more results than \( (a) \rightarrow (b:Person) \), a right outer join — indicated by NodeRightOuterHashJoin — is planned instead.

**Query**

```cypher
MATCH (a:Person)
OPTIONAL MATCH (a)--> (b:Person)
USING JOIN ON a
RETURN a.name, b.name
```
29.3.56. 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**

```cypher
MATCH (me:Person)-[:FRIENDS_WITH]-(a)-[:FRIENDS_WITH]-(other)
WHERE NOT (me)-[:FRIENDS_WITH]-(other)
RETURN other.name
```
29.3.57. Triadic Build

The TriadicBuild operator is used in conjunction with TriadicFilter to solve triangular queries, such as the very common ‘find my friend-of-friends that are not already my friend’. These two operators are specific to Pipelined runtime and together perform the same logic as TriadicSelection does for other runtimes. TriadicBuild builds a set of all friends, which is later used by TriadicFilter. The example finds the names of all friends of my friends that are not already my friends.

Query

```cypher
MATCH (me:Person)-[:FRIENDS_WITH]-()-[:FRIENDS_WITH]-(other)
WHERE NOT (me)-[:FRIENDS_WITH]-()-(other)
RETURN other.name
```
29.3.58. Triadic Filter

The `TriadicFilter` operator is used in conjunction with `TriadicBuild` to solve triangular queries, such as the very common ‘find my friend-of-friends that are not already my friend’. These two operators are specific to Pipelined runtime and together perform the same logic as `TriadicSelection` does for other runtimes. `TriadicFilter` uses a set of friends previously built by `TriadicBuild` 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]-(other)
WHERE NOT (me)-[:FRIENDS_WITH]-(other)
RETURN other.name
```

Query Plan

Compiler CYPHER 4.4
Planner COST
Runtime PIPELINED
Runtime version 4.4

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Memory (Bytes)</th>
<th>Page Cache Hits/Misses</th>
<th>Time (ms)</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>'other.name'</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0/0</td>
<td>5.796</td>
<td>In Pipeline 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>2/0</td>
<td>0.686</td>
<td>6984</td>
<td>In Pipeline 3</td>
</tr>
<tr>
<td>+Projection</td>
<td>other.name AS 'other.name'</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0/0</td>
<td>0.673</td>
<td>Fused in Pipeline 2</td>
</tr>
<tr>
<td>+TriadicFilter</td>
<td>WHERE NOT (me)--(other)</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0/0</td>
<td>4.635</td>
<td>In Pipeline 3</td>
</tr>
<tr>
<td>+Apply</td>
<td></td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0/0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+Filter</td>
<td>not anon_2 = anon_4</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0/0</td>
<td>0.673</td>
</tr>
<tr>
<td></td>
<td>+Expand(All)</td>
<td>(anon_3)-[anon_4:FRIENDS_WITH]-(other)</td>
<td>0</td>
<td>6</td>
<td>14</td>
<td>0</td>
<td>0/0</td>
<td>18.689</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+Argument</td>
<td>anon_3, anon_2</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0/0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+TriadicBuild</td>
<td>(me)--(anon_3)</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0/0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+Expand(All)</td>
<td>(me)-[anon_2:FRIENDS_WITH]-(anon_3)</td>
<td>4</td>
<td>4</td>
<td>19</td>
<td>0</td>
<td>0/0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>+Filter</td>
<td>me:Person</td>
<td>14</td>
<td>14</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>+AllNodesScan</td>
<td>me</td>
<td>35</td>
<td>35</td>
<td>36</td>
<td>2/0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>35</td>
<td>35</td>
<td>36</td>
<td>2/0</td>
</tr>
</tbody>
</table>

Total database accesses: 73, total allocated memory: 7248
29.3.59. 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 4.4
Planner COST
Runtime PIPELINED
Runtime version 4.4

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Memory (Bytes)</th>
<th>Page Cache Hits/Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>p, t</td>
<td>140</td>
<td>140</td>
<td>0</td>
<td></td>
<td>2/0</td>
</tr>
<tr>
<td>Time (ms)</td>
<td>In Pipeline 2</td>
<td>3.517</td>
<td>In Pipeline 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+CartesianProduct</td>
<td></td>
<td>140</td>
<td>140</td>
<td>0</td>
<td>3656</td>
</tr>
<tr>
<td>Time (ms)</td>
<td>In Pipeline 2</td>
<td>0.901</td>
<td>In Pipeline 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+Filter</td>
<td>t:Team</td>
<td>10</td>
<td>10</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Time (ms)</td>
<td>Fused in Pipeline 1</td>
<td>0.157</td>
<td>Fused in Pipeline 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>+AllNodesScan</td>
<td>t</td>
<td>35</td>
<td>35</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>Time (ms)</td>
<td>Fused in Pipeline 1</td>
<td>0.157</td>
<td>Fused in Pipeline 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>+Filter</td>
<td>p:Person</td>
<td>14</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Time (ms)</td>
<td>Fused in Pipeline 0</td>
<td>0.347</td>
<td>Fused in Pipeline 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>+AllNodesScan</td>
<td>p</td>
<td>35</td>
<td>35</td>
<td>36</td>
</tr>
<tr>
<td>Time (ms)</td>
<td>Fused in Pipeline 0</td>
<td>0.347</td>
<td>Fused in Pipeline 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total database accesses: 72, total allocated memory: 3736</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

29.3.60. 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.
29.3.61. 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

```cypher
MATCH (a)-[r]->(b) DELETE r, a, b
```

Total database accesses: 9, total allocated memory: 64
### 29.3.62. 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 4.4
Planner COST
Runtime PIPELINED
Runtime version 4.4

| Operator          | Details                                              | Estimated Rows | Rows | DB |
|-------------------|------------------------------------------------------|----------------+------+----|
| +ProduceResults   | location, people                                     |              4 |    6 |    |
|                   |                                                     | 0/0 | 0.293 | In Pipeline 1 |
|                   |                                                     | +------------------------------------------------------+----------------+------|
| +EagerAggregation | cache[l.name] AS location, collect(p.name) AS people |              4 |    6 |    |
|                   |                                                     | 30 | 3168 |                |
|                   |                                                     | +------------------------------------------------------+----------------+------|
| +Filter           | p:Person                                             |             15 |   15 |    |
|                   |                                                     | 15 |    | Fused in Pipeline 0 |
|                   |                                                     | +------------------------------------------------------+----------------+------|
| +Expand(All)      | (l)<-[anon_0:WORKS_IN]-(p)                           |             15 |   15 |    |
|                   |                                                     | 16 |    | Fused in Pipeline 0 |
|                   |                                                     | +------------------------------------------------------+----------------+------|
| +CacheProperties  | cache[l.name]                                        |             10 |   10 |    |
|                   |                                                     | 10 |    | Fused in Pipeline 0 |
|                   |                                                     | +------------------------------------------------------+----------------+------|
| +Filter           | l:Location                                           |             10 |   10 |    |
|                   |                                                     | 0 |    | Fused in Pipeline 0 |
|                   |                                                     | +------------------------------------------------------+----------------+------|
| +AllNodesScan     | l                                                    |             35 |   35 |    |
|                   |                                                     | 36 | 112 | 21.472 | Fused in Pipeline 0 |
|                   |                                                     | +------------------------------------------------------+----------------+------|

Total database accesses: 107, total allocated memory: 3248
```

29.3.63. Ordered Aggregation

The OrderedAggregation operator is an optimization of the EagerAggregation operator that takes advantage of the ordering of the incoming rows. This operator uses lazy evaluation and has a lower memory pressure in the system than the EagerAggregation operator.

Query

```
MATCH (p:Person) WHERE p.name STARTS WITH 'P' RETURN p.name, count(*) AS count
```
### Query Plan

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimated Rows</td>
<td>Rows</td>
</tr>
<tr>
<td>+ProduceResults</td>
<td><code>p.name</code>, count</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>+OrderedAggregation</td>
<td>cache[p.name] AS <code>p.name</code>, count(*) AS count</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>+NodeIndexSeekByRange</td>
<td>BTREE INDEX p:Person(name) WHERE name STARTS WITH $autostring_0, cache[p.name]</td>
</tr>
</tbody>
</table>

Total database accesses: 3, total allocated memory: 1896

### 29.3.64. 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**

```cypher
MATCH (p:Person) RETURN count(p) AS people
```
29.3.65. 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
```
29.3.66. 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**

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

**Query Plan**

```
+-----------------+----------------------------+----------------+------+---------+----------------
| Operator        | Details                    | Estimated Rows | Rows | DB Hits | Memory (Bytes) |
| Cache Hits/Misses | Time (ms) | Other               | +-----------------+----------------------------+----------------+------+---------+----------------
| +ProduceResults | l                          |             14 |    6 |       0 |                |
| | Fused in Pipeline 0 |            +-----------------+----------------------------+----------------+------+---------+----------------
| +Distinct       | l                          |             14 |    6 |       0 |            224 |
| | Fused in Pipeline 0 |            +-----------------+----------------------------+----------------+------+---------+----------------
| +Filter         | p:Person                   |             15 |   15 |      15 |                |
| | Fused in Pipeline 0 |            +-----------------+----------------------------+----------------+------+---------+----------------
| +Expand(All)    | (l)<-[anon_0:WORKS_IN]-|             15 |   15 |      16 |                |
| | Fused in Pipeline 0 |            +-----------------+----------------------------+----------------+------+---------+----------------
| +Filter         | l:Location                 |             10 |   10 |       0 |                |
| | Fused in Pipeline 0 |            +-----------------+----------------------------+----------------+------+---------+----------------
| +AllNodesScan   | l                          |             35 |   35 |      36 |            112 |
5/0 | 1.025 | Fused in Pipeline 0 |            +-----------------+----------------------------+----------------+------+---------+----------------
```

Total database accesses: 67, total allocated memory: 304

29.3.67. Ordered Distinct

The **OrderedDistinct** operator is an optimization of the **Distinct** operator that takes advantage of the ordering of the incoming rows. This operator has a lower memory pressure in the system than the **Distinct** operator.

**Query**

```cypher
MATCH (p:Person) WHERE p.name STARTS WITH 'P' RETURN DISTINCT p.name
```

706
### Query Plan

**Compiler** CYPER 4.4  
**Planner** COST  
**Runtime** PIPELINED  
**Runtime version** 4.4

| Operator              | Details                                                                        | Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Ordered by | Other            |
|-----------------------+--------------------------------------------------------------------------------|----------------+------+---------+----------------+------------------------+-----------+------------+-------------------|
| +ProduceResults       | `p.name`                                                                       | 0               | 2    | 0       |                | 0/0                   | 0.083     | p.name ASC | In Pipeline 0     |
| +OrderedDistinct      | cache[p.name] AS `p.name`                                                      | 0               | 2    | 0       | 32             | 0/0                   | 4.215     | p.name ASC | In Pipeline 0     |
| NodeIndexSeekByRange  | BTREE INDEX p:Person(name) WHERE name STARTS WITH $autostring_0, cache[p.name] | 0               | 2    | 3       | 112            | 0/1                   | 0.314     | p.name ASC | In Pipeline 0     |

Total database accesses: 3, total allocated memory: 176

### 29.3.68. 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
```
29.3.69. Limit

The **Limit** operator returns the first 'n' rows from the incoming input.

**Query**

```cypher
MATCH (p:Person) RETURN p LIMIT 3
```
Query Plan

Compiler CYpher 4.4
Planner COST
Runtime PIPELINED
Runtime version 4.4

+-----------------+----------+----------------+------+---------+----------------+------------------------+
| Operator         | Details  | Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses |
| Time (ms)        | Other    |                |      |         |                |                        |
+-----------------+----------+----------------+------+---------+----------------+------------------------+
| +ProduceResults  | p        |              3  |    3 |       0 |                |                        |
| Fused in Pipeline 0 |       |               +----------+----------------+------+---------+----------------+                        |
+---------------------+
| +Limit              | 3        |              3  |    3 |       0 |             32 |                        |
| Fused in Pipeline 0 |       |               +----------+----------------+------+---------+----------------+                        |
+---------------------+
| +Filter             | p:Person |              3  |    3 |       0 |                |                        |
| Fused in Pipeline 0 |       |               +----------+----------------+------+---------+----------------+                        |
+---------------------+
| +AllNodesScan      | p        |              8  |    4 |       5 |            112 |                    3/0 |
| 1.107              | Fused in Pipeline 0 |       |               +----------+----------------+------+---------+----------------+-------------------------------------------------------------|
+-----------------+----------+----------------+------+---------+----------------+------------------------+

Total database accesses: 5, total allocated memory: 176

29.3.70. Skip

The **Skip** operator skips 'n' rows from the incoming rows.

Query

```cypher
MATCH (p:Person)
RETURN p
ORDER BY p.id
SKIP 1
```
<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Memory (Bytes)</th>
<th>Page Cache Hits/Misses</th>
<th>Time (ms)</th>
<th>Ordered by</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>p</td>
<td>13</td>
<td>13</td>
<td>0</td>
<td></td>
<td></td>
<td>0.468</td>
<td>p.id ASC</td>
<td>In Pipeline 1</td>
</tr>
<tr>
<td>+Skip</td>
<td>$autoint_0</td>
<td>13</td>
<td>13</td>
<td>0</td>
<td>32</td>
<td></td>
<td>1.397</td>
<td>p.id ASC</td>
<td>In Pipeline 1</td>
</tr>
<tr>
<td>+Sort</td>
<td><code>p.id</code> ASC</td>
<td>14</td>
<td>14</td>
<td>0</td>
<td>1800</td>
<td></td>
<td>0.262</td>
<td>p.id ASC</td>
<td>In Pipeline 1</td>
</tr>
<tr>
<td>+Projection</td>
<td>p.id AS <code>p.id</code></td>
<td>14</td>
<td>14</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fused in Pipeline 0</td>
</tr>
<tr>
<td>+Filter</td>
<td>p:Person</td>
<td>14</td>
<td>14</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fused in Pipeline 0</td>
</tr>
<tr>
<td>+AllNodesScan</td>
<td>p</td>
<td>35</td>
<td>35</td>
<td>36</td>
<td>112</td>
<td></td>
<td>0.328</td>
<td></td>
<td>Fused in Pipeline 0</td>
</tr>
</tbody>
</table>

Total database accesses: 36, total allocated memory: 1912

### 29.3.71. 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**

```cypher
MATCH (p:Person) RETURN p ORDER BY p.name
```
29.3.72. Partial Sort

The PartialSort operator is an optimization of the Sort operator that takes advantage of the ordering of the incoming rows. This operator uses lazy evaluation and has a lower memory pressure in the system than the Sort operator. Partial sort is only applicable when sorting on multiple columns.

Query

MATCH (p:Person) WHERE p.name STARTS WITH 'P' RETURN p ORDER BY p.name, p.age
### Query Plan

Compiler CYpher 4.4  
Planner COST  
Runtime PIPELINED  

Runtime version 4.4

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimated Rows</td>
<td>Rows</td>
</tr>
<tr>
<td>Other</td>
<td></td>
</tr>
</tbody>
</table>

| +ProduceResults   | p                                                                                     |
| 0 | 2 | 0 | 0 | 2/0 | 0.157 | p.name ASC, p.age ASC | In Pipeline 1 |

| +PartialSort     | p.name ASC, p.age ASC                                                                  |
| 0 | 2 | 0 | 3096 | 0/0 | 0.248 | p.name ASC, p.age ASC | In Pipeline 1 |

| 0 | 2 | 0 | 0 |   |     | p.name ASC | Fused in Pipeline 0 |

| +NodeIndexSeekByRange | BTREE INDEX p:Person(name) WHERE name STARTS WITH $autostring_0, cache[p.name] |
| 0 | 2 | 3 | 112 | 0/1 | 0.859 | p.name ASC | Fused in Pipeline 0 |

Total database accesses: 3, total allocated memory: 3160

### 29.3.73. 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**

```cypher
MATCH (p:Person) RETURN p ORDER BY p.name LIMIT 2
```
29.3.74. Partial Top

The `PartialTop` operator is an optimization of the `Top` operator that takes advantage of the ordering of the incoming rows. This operator uses lazy evaluation and has a lower memory pressure in the system than the `Top` operator. Partial top is only applicable when sorting on multiple columns.

Query

```
MATCH (p:Person) WHERE p.name STARTS WITH 'P' RETURN p ORDER BY p.name, p.age LIMIT 2
```
### Query Plan

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Other</td>
<td>+------------------------------+---------------------------------------------+------------------------+-----------</td>
</tr>
<tr>
<td>+ProduceResults</td>
<td>p</td>
</tr>
<tr>
<td>Estimated Rows</td>
<td>Rows</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>+PartialTop</td>
<td><code>p.name</code> ASC, <code>p.age</code> ASC LIMIT 2</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>+NodeIndexSeekByRange</td>
<td>BTREE INDEX p:Person(name) WHERE name STARTS WITH $autostring_0, cache[p.name]</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

**Total database accesses:** 3, **total allocated memory:** 696

### 29.3.75. Union

The **Union** operator concatenates the results from the right child operator with the results from the left child operator.

**Query**

```cypher
MATCH (p:Location)
RETURN p.name
UNION ALL
MATCH (p:Country)
RETURN p.name
```
29.3.76. Unwind

The **Unwind** operator returns one row per item in a list.

**Query**

```
UNWIND range(1, 5) as value return value
```
29.3.77. Exhaustive Limit

The ExhaustiveLimit operator is just like a normal Limit but will always exhaust the input. Used when combining LIMIT and updates

Query

```
MATCH (p:Person) SET p.seen=true RETURN p LIMIT 3
```
## Query Plan

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
<th>Estimated Rows</th>
<th>DB Hits</th>
<th>Memory (Bytes)</th>
<th>Page Cache Hits/Misses</th>
<th>Time (ms)</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProduceResults</td>
<td>p</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fused in Pipeline 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ExhaustiveLimit</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fused in Pipeline 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SetProperty</td>
<td>p.seen = true</td>
<td>14</td>
<td>14</td>
<td>28</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fused in Pipeline 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Filter</td>
<td>p:Person</td>
<td>14</td>
<td>14</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fused in Pipeline 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AllNodesScan</td>
<td>p</td>
<td>35</td>
<td>35</td>
<td>36</td>
<td>112</td>
<td>3/0</td>
<td>3.065</td>
</tr>
<tr>
<td></td>
<td>Fused in Pipeline 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total database accesses: 64, total allocated memory: 192

### 29.3.78. 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 4.4  
Planner COST  
Runtime PIPELINED  
Runtime version 4.4

| Operator         | Details                                               | Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Other         |
|------------------|-------------------------------------------------------+----------------+------+---------+----------------+------------------------+-----------+---------------|
| +ProduceResults  | p, q                                                  | 1              | 1    | 0       | 0 |                | 2/0 | 0.197 | In Pipeline 2 |
| |                +-------------------------------------------------------+----------------+------+---------+----------------+------------------------+-----------+---------------|
| +Apply           |                                                       | 1              | 1    | 0       | 0 |                | 0/0 | 0.022 |               |
| | \               +-------------------------------------------------------+----------------+------+---------+----------------+------------------------+-----------+---------------|
| | | +Optional      | p                                                     | 1              | 1    | 0       | 768 |                | 0/0 | 0.874 | In Pipeline 2 |
| | | |              +-------------------------------------------------------+----------------+------+---------+----------------+------------------------+-----------+---------------|
| | | +NodeIndexSeek | BTREE INDEX q:Person(name) WHERE name = $autostring_1 | 1              | 0    | 1       | 2152 |                | 1/0 | 1.376 | In Pipeline 1 |
| | | |              +-------------------------------------------------------+----------------+------+---------+----------------+------------------------+-----------+---------------|
| | | +NodeIndexSeek | BTREE INDEX p:Person(name) WHERE name = $autostring_0 | 1              | 1    | 2       | 112  |                | 0/1 | 0.566 | In Pipeline 0 |
| | | |              +-------------------------------------------------------+----------------+------+---------+----------------+------------------------+-----------+---------------|

Total database accesses: 3, total allocated memory: 3000

### 29.3.79. Project Endpoints

The **ProjectEndpoints** operator projects the start and end node of a relationship.

**Query**

```cypher
CREATE (n:Person)-[:KNOWS]->(m) WITH p AS r MATCH (u)-[r]->(v) RETURN u, v
```
### Query Plan

**Compiler CYPER 4.4**

**Planner COST**

**Runtime PIPELINED**

**Runtime version 4.4**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Memory (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>u, v</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fused in Pipeline 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Apply</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fused in Pipeline 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ProjectEndpoints</td>
<td>(u)-[r*]-&gt;(v)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fused in Pipeline 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Argument</td>
<td>r</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>4200</td>
</tr>
<tr>
<td></td>
<td>Fused in Pipeline 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Projection</td>
<td>p AS r</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fused in Pipeline 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Create</td>
<td>(n), (m), (n)-[p:KNOWS]-&gt;(m)</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fused in Pipeline 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total database accesses: 4, total allocated memory: 4280

### 29.3.80. 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 4.4  
Planner COST  
Runtime PIPELINED  
Runtime version 4.4

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Page Cache Hits/Misses</th>
<th>Time (ms)</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>greeting</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>4.747</td>
<td>In Pipeline 1</td>
</tr>
<tr>
<td>+Projection</td>
<td>$autostring_0 AS greeting</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0/0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total database accesses: 0, total allocated memory: 176

### 29.3.81. Shortest path

The **ShortestPath** operator finds one or all shortest paths between two previously matches node variables.

**Query**

```cypher
MATCH (andy:Person {name: 'Andy'}),(mattias:Person {name: 'Mattias'}),
p = shortestPath((andy)-[*]-(mattias))
RETURN p
```

### Query Plan

Compiler CYPHER 4.4  
Planner COST  
Runtime PIPELINED  
Runtime version 4.4

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Memory (Bytes)</th>
<th>Page Cache Hits/Misses</th>
<th>Time (ms)</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>p</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td>4.747</td>
<td>In Pipeline 1</td>
</tr>
<tr>
<td>+ShortestPath</td>
<td>p = (andy)-[anon_0*]-(mattias)</td>
<td>1</td>
<td>1</td>
<td>3176</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+MultiNodeIndexSeek</td>
<td>BTREE INDEX andy:Person(name) WHERE name = $autostring_0</td>
<td>1</td>
<td>1</td>
<td>112</td>
<td>1/1</td>
<td>1.937</td>
<td>In Pipeline 0</td>
<td></td>
</tr>
</tbody>
</table>

Total database accesses: 5, total allocated memory: 3240
29.3.82. Empty Row

The **EmptyRow** operator returns a single row with no columns.

**Query**

```
CYPHER runtime=slopped FOREACH (value IN [1,2,3] |
MERGE (:Person {age: value})
)
```

**Query Plan**

```
| Operator        | Details                              | Estimated Rows | Rows | DB Hits | Page Cache Hits/Misses |
|-----------------+--------------------------------------+----------------+------+---------+------------------------|
| +ProduceResults |                                      | 1              | 0    | 0       | 0/0                    |
| |               +--------------------------------------+----------------+------+---------+------------------------|
| +EmptyResult    |                                      | 1              | 0    | 0       | 0/0                    |
| |               +--------------------------------------+----------------+------+---------+------------------------|
| +Foreach        | value IN [1, 2, 3]                     | 1              | 1    | 0       | 0/0                    |
| | |
| | +Merge        | CREATE (anon_0:Person {age: value})    | 1              | 3    | 9       | 0/0                    |
| | |             +--------------------------------------+----------------+------+---------+------------------------|
| | +Filter       | anon_0:Person AND anon_0.age = value   | 1              | 0    | 184     | 2/0                    |
| | |             +--------------------------------------+----------------+------+---------+------------------------|
| | +AllNodesScan | anon_0                                  | 35             | 108  | 111     | 3/0                    |
| | |               +--------------------------------------+----------------+------+---------+------------------------|
| +EmptyRow       |                                      | 1              | 1    | 0       | 0/0                    |
|                  +--------------------------------------+----------------+------+---------+------------------------|
```

Total database accesses: 304, total allocated memory: 64

29.3.83. Procedure Call

The **ProcedureCall** operator indicates an invocation to a procedure.

**Query**

```
CALL db.labels() YIELD label RETURN * ORDER BY label
```
### Query Plan

**Compiler CYPER 4.4**

**Planner COST**

**Runtime PIPELINED**

**Runtime version 4.4**

| Operator        | Details                           | Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Ordered by | Other               |
|-----------------|-----------------------------------|----------------+------+---------+----------------+------------------------+------------+-------------+---------------------|
| +ProduceResults | label                             | 10             | 4    | 0       |                |                        | 0.280      | label ASC   | In Pipeline 1       |
| +Sort           | label ASC                         | 10             | 4    | 0       | 1064           |                        | 5.061      | label ASC   | In Pipeline 1       |
| +ProcedureCall  | db.labels() :: (label :: STRING?) | 10             | 4    |         |                |                        |            |             | Fused in Pipeline 0 |

Total database accesses: 7, total allocated memory: 1128

### 29.3.84. Cache Properties

The CacheProperties operator reads nodes and relationship properties and caches them in the current row. Future accesses to these properties can avoid reading from the store which will speed up the query. In the plan below we will cache `l.name` before `Expand(All)` where there are fewer rows.

**Query**

```
MATCH (l:Location)<-[::WORKS_IN]-(p:Person) RETURN l.name AS location, p.name AS name
```
### 29.3.85. Create Nodes / Relationships

The **Create** operator is used to create nodes and relationships.

**Query**

```
CREATE (max:Person {name: 'Max'}), (chris:Person {name: 'Chris'})
CREATE (max)-[:FRIENDS_WITH]->(chris)
```
29.3.86. Delete

The Delete operator is used to delete a node or a relationship.

Query

```cypher
MATCH (me:Person {name: 'me'})-[w:WORKS_IN {duration: 190}]->(london:Location {name: 'London'})
DELETE w
```
### Query Plan

**Compiler** CYpher 4.4  
**Planner** COST  
**Runtime** PIPELINED  
**Runtime version** 4.4

| Operator        | Details                           | Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Other                         |
|-----------------+----------------------------------+----------------+------+---------+----------------+------------------------+-----------+-------------------------------|
| +ProduceResults |                                  | 0              | 0    | 0       |                |                        |           | Fused in Pipeline 1            |
| |                  +-------------------------------------------------------------+-----------|-------------------------------|
| +EmptyResult    |                                  | 0              | 0    | 0       |                |                        |           | Fused in Pipeline 1            |
| |                  +-------------------------------------------------------------+-----------|-------------------------------|
| +Delete         | w                                | 0              | 1    | 1       |                |                        |           | Fused in Pipeline 1            |
| |                  +-------------------------------------------------------------+-----------|-------------------------------|
| +Eager          | delete overlap: w                 | 0              | 1    | 0       | 3192           | 1/0                    | 1.753     | Fused in Pipeline 1            |
| |                  +-------------------------------------------------------------+-----------|-------------------------------|
| +Filter         | london.name = $autostring_2 AND w.duration = $autoint_1 AND london:Location | 0          | 1    | 4       |                |                        |           | Fused in Pipeline 0            |
| |                  +-------------------------------------------------------------+-----------|-------------------------------|
| +Expand(All)    | (me)-[w:WORKS_IN]->(london)       | 1              | 1    | 3       |                |                        |           | Fused in Pipeline 0            |
| |                  +-------------------------------------------------------------+-----------|-------------------------------|
| +NodeIndexSeek  | BTREE INDEX me:Person(name) WHERE name = $autostring_0               | 1              | 1    | 2       | 112            | 4/1                    | 1.110     | Fused in Pipeline 0            |
| |                  +-------------------------------------------------------------+-----------|-------------------------------|

Total database accesses: 10, total allocated memory: 3272

---

### 29.3.87. Detach Delete

The **DetachDelete** operator is used in all queries containing the **DETACH DELETE** clause, when deleting nodes and their relationships.

**Query**

```cypher
MATCH (p:Person)
DETACH DELETE p
```
### Query Plan

Compiler CYPHER 4.4  
Planner COST  
Runtime PIPELINED  
Runtime version 4.4  

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Memory (Bytes)</th>
<th>Page Cache Hits/Misses</th>
<th>Time (ms)</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td></td>
<td>14</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fused in Pipeline 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+EmptyResult</td>
<td></td>
<td>14</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fused in Pipeline 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+DetachDelete</td>
<td>p</td>
<td>14</td>
<td>14</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fused in Pipeline 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Filter</td>
<td>p:Person</td>
<td>14</td>
<td>14</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fused in Pipeline 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+AllNodesScan</td>
<td>p</td>
<td>35</td>
<td>35</td>
<td>36</td>
<td>112</td>
<td>19/0</td>
<td>3.102</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fused in Pipeline 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total database accesses: 67, total allocated memory: 192

### 29.3.88. Set Labels

The `SetLabels` operator is used when setting labels on a node.

**Query**

```cypher
MATCH (n)
SET n:Person
```
29.3.89. Remove Labels

The **RemoveLabels** operator is used when deleting labels from a node.

**Query**

```
MATCH (n)
REMOVE n:Person
```
29.3.90. Set Node Properties From Map

The `SetNodePropertiesFromMap` operator is used when setting properties from a map on a node.

**Query**

```cypher
MATCH (n)
SET n = {weekday: 'Monday', meal: 'Lunch'}
```
29.3.91. Set Relationship Properties From Map

The `SetRelationshipPropertiesFromMap` operator is used when setting properties from a map on a relationship.

Query

```
MATCH (n)-[r]->(m)
SET r = {weight: 5, unit: 'kg'}
```
### 29.3.92. Set Property

The **SetProperty** operator is used when setting a property on a node or relationship.

**Query**

```cypher
MATCH (n)
SET n.checked = true
```
29.3.93. Create Unique Constraint

The `CreateUniqueConstraint` operator creates a unique constraint on a set of properties for all nodes having a certain label. The following query will create a unique constraint with the name `uniqueness` on the `name` property of nodes with the `Country` label.

**Query**

```
CREATE CONSTRAINT uniqueness FOR (c:Country) REQUIRE c.name is UNIQUE
```

**Query Plan**

`Compiler CYpher 4.4`

`Planner ADMINISTRATION`

`Runtime SCHEMA`

Runtime version 4.4

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>+CreateConstraint</td>
<td>CONSTRAINT uniqueness FOR (c:Country) REQUIRE (c.name) IS UNIQUE</td>
</tr>
</tbody>
</table>

Total database accesses: 106, total allocated memory: 176
29.3.94. Drop Unique Constraint == Deprecated

The `DropUniqueConstraint` operator removes a unique constraint from all nodes having a certain set of properties and label. The following query will drop a unique constraint on the `name` property of nodes with the `Country` label.

Query

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

Query Plan

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>+DropConstraint</td>
<td>CONSTRAINT ON (c:Country) ASSERT (c.name) IS UNIQUE</td>
</tr>
</tbody>
</table>

Total database accesses: ?

29.3.95. Create Constraint only if it does not already exist

To not get an error creating the same constraint twice, we use the `DoNothingIfExists` operator for constraints. This will make sure no other constraint with the given name or another constraint of the same type and schema already exists before the specific `CreateConstraint` operator creates the constraint. If it finds a constraint with the given name or with the same type and schema it will stop the execution and no new constraint is created. The following query will create a unique constraint with the name `uniqueness` on the `name` property of nodes with the `Country` label only if no constraint named `uniqueness` or unique constraint on (:Country {name}) already exists.

Query

```cypher
CREATE CONSTRAINT uniqueness IF NOT EXISTS FOR (c:Country) REQUIRE c.name IS UNIQUE
```
29.3.96. Create Node Property Existence Constraint

The `CREATE CONSTRAINT` operator creates an existence constraint with the name `existence` on a property for all nodes having a certain label. This will only appear in Enterprise Edition.

**Query**

```cypher
CREATE CONSTRAINT existence FOR (p:Person) REQUIRE p.name IS NOT NULL
```

**Query Plan**

Compiler: CYPHER 4.4  
Planner: ADMINISTRATION  
Runtime: SCHEMA  
Runtime version: 4.4

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>+CreateConstraint</td>
<td>CONSTRAINT existence FOR (p:Person) REQUIRE (p.name) IS NOT NULL</td>
</tr>
</tbody>
</table>

Total database accesses: ?

29.3.97. Drop Node Property Existence Constraint

The `DROP CONSTRAINT` operator removes an existence constraint from a property for all nodes having a certain label. This will only appear in Enterprise Edition.

**Query**

```cypher
DROP CONSTRAINT ON (p:Person) ASSERT exists(p.name)
```

**Query Plan**

Compiler: CYPHER 4.4  
Planner: ADMINISTRATION  
Runtime: SCHEMA  
Runtime version: 4.4

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>+CreateConstraint</td>
<td>CONSTRAINT existence FOR (p:Person) REQUIRE (p.name) IS NOT NULL</td>
</tr>
</tbody>
</table>

Total database accesses: ?
29.3.98. Create Node Key Constraint

The `CreateNodeKeyConstraint` operator creates a node key constraint with the name `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 node_key FOR (e:Employee) REQUIRE (e.firstname, e.surname) IS NODE KEY
```

29.3.99. Drop Node Key Constraint == Deprecated

The `DropNodeKeyConstraint` operator removes a node key constraint 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
```
29.3.100. Create Relationship Property Existence Constraint

The `CreateRelationshipPropertyExistenceConstraint` operator creates an existence constraint with the name `existence` on a property for all relationships of a certain type. This will only appear in Enterprise Edition.

Query

```
CREATE CONSTRAINT existence FOR ()-[l:LIKED]-() REQUIRE l.when IS NOT NULL
```

Query Plan

```
+-----------------+------------------------------------------------------------------------+
| Operator        | Details                                                                |
+-----------------+------------------------------------------------------------------------+
| +CreateConstraint | CONSTRAINT existence FOR ()-[l:LIKED]-() REQUIRE (l.when) IS NOT NULL |
+-----------------+------------------------------------------------------------------------+
Total database accesses: ?
```

29.3.101. 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

```
+-----------------+------------------------------------------------------------------------+
| Operator        | Details                                                                |
+-----------------+------------------------------------------------------------------------+
| -DropConstraint | CONSTRAINT ON (e:Employee) ASSERT (e.firstname, e.surname) IS NODE KEY |
+-----------------+------------------------------------------------------------------------+
Total database accesses: ?
```
29.3.102. Drop Constraint by name

The **DropConstraint** operator removes a constraint using the name of the constraint, no matter the type.

**Query**

```
DROP CONSTRAINT name
```

**Query Plan**

```
Compiler CYpher 4.4
Planner ADMINISTRATION
Runtime SCHEMA
Runtime version 4.4
+-----------------+-----------------+
| Operator        | Details         |
+-----------------+-----------------+
| +DropConstraint | CONSTRAINT name |
+-----------------+-----------------+
Total database accesses: ?
```

29.3.103. Listing constraints

The **ShowConstraints** operator lists constraints. It may include filtering on constraint type and can have either default or full output.

**Query**

```
SHOW CONSTRAINTS
```
29.3.104. Create Index

The `CreateIndex` operator creates an index. This index can either be a b-tree, fulltext, text, or token lookup index. The following query will create an index with the name `my_index` on the `name` property of nodes with the `Country` label.

**Query**

```cypher
CREATE INDEX my_index FOR (c:Country) ON (c.name)
```

**Query Plan**

```
+--------------+-----------------------------------------------+
| Operator     | Details                                       |
+--------------+-----------------------------------------------+
| +CreateIndex | BTREE INDEX my_index FOR (:Country) ON (name) |
+--------------+-----------------------------------------------+
```

Total database accesses: 1, total allocated memory: 64

29.3.105. Create Index only if it does not already exist

To not get an error creating the same index twice, we use the `DoNothingIfExists` operator for indexes. This will make sure no other index with the given name or schema already exists before the `CreateIndex` operator creates an index. If it finds an index with the given name or schema it will stop the execution and no new index is created. The following query will create an index with the name `my_index` on the `since` property of relationships with the `KNOWS` relationship type only if no such index already exists.

...
CREATE INDEX my_index IF NOT EXISTS FOR ()-[k:KNOWS]-() ON (k.since)

29.3.106. Drop Index by schema == Deprecated

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)

29.3.107. Drop Index by name

The DropIndex operator removes an index using the name of the index.

Query

DROP INDEX name
29.3.108. Listing indexes

The `ShowIndexes` operator lists indexes. It may include filtering on index type and can have either default or full output.

Query

```
SHOW INDEXES
```

29.3.109. Listing functions

The `ShowFunctions` operator lists functions. It may include filtering on built-in vs user-defined functions as well as if a given user can execute the function. The output can either be default or full output.
29.3.110. Listing procedures

The ShowProcedures operator lists procedures. It may include filtering on whether a given user can execute the procedure and can have either default or full output.

Query

SHOW PROCEDURES

Query Plan

Compiler CYPHER 4.4
Planner COST
Runtime SLOTTED
Runtime version 4.4

| Operator        | Details                                             | Estimated Rows | Rows | DB Hits | Page Cache Hits/Misses |
|-----------------+-----------------------------------------------------|----------------+------|--------+-----------------------|
| +ProduceResults | name, description, mode, worksOnSystem             | 10             | 73    | 0      | 0/0                   |
| |               +----------------------------------------|----------------+------|--------+-----------------------|
| +ShowProcedures | proceduresForUser(all), defaultColumns             | 10             | 73    | 0      | 0/0                   |

Total database accesses: 0, total allocated memory: 64
29.3.111. Listing transactions

The `ShowTransactions` operator lists transactions. It may include filtering on given ids and can have either default or full output.

**Query**

```
SHOW TRANSACTIONS
```

**Query Plan**

<table>
<thead>
<tr>
<th>Compiler CYPHER 4.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Planner COST</td>
</tr>
<tr>
<td>Runtime SLOTTED</td>
</tr>
<tr>
<td>Runtime version 4.4</td>
</tr>
</tbody>
</table>

**View**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>database, transactionId, currentQueryId, connectionId, clientAddress, username, currentQuery, startTime, status, elapsedTime, allocatedBytes</td>
</tr>
<tr>
<td>+ShowTransactions</td>
<td>defaultColumns, allTransactions</td>
</tr>
</tbody>
</table>

Total database accesses: 0, total allocated memory: 64

29.3.112. Terminating transactions

The `TerminateTransactions` operator terminates transactions by ID.

**Query**

```
TERMINATE TRANSACTIONS 'database-transaction-123'
```
### 29.4. 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 `name='Kevin Bacon'` — 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

29.4.1. Shortest path with fast algorithm

Query

```
MATCH (KevinB:Person {name: 'Kevin Bacon'}),
(Al:Person {name: 'Al Pacino'}),
p = shortestPath((KevinB)-[:ACTED_IN*]-(Al))
WHERE all(r IN relationships(p) WHERE r.role IS NOT NULL)
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 4.4
Planner COST
Runtime PIPELINED
Runtime version 4.4
+---------------------+---------------------------------+-------+----------------+------------------------+-----------+---------------+
| Operator             | Details                         | Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Other         |
+---------------------+---------------------------------+-------+----------------+------------------------+-----------+---------------+
| +ProduceResults      | p                               |              2 |    1 |       0 |                |                    1/0 |     0.222 | In Pipeline 1 |
| +ShortestPath        | p = (KevinB)-[anon_0:ACTED_IN*]-(Al) WHERE all(r IN relationships(p) WHERE r.role IS NOT NULL) |              2 |    1 |      23 |           3176 |                        |           | In Pipeline 1 |
| +MultiNodeIndexSeek  | BTREE INDEX KevinB:Person(name) WHERE name = $autostring_0, |              2 |    1 |       4 |            112 |                    1/1 |     0.483 | In Pipeline 0 |
|                     | BTREE INDEX Al:Person(name) WHERE name = $autostring_1 | | | | | | |
+---------------------+---------------------------------+-------+----------------+------------------------+-----------+---------------+
Total database accesses: 27, total allocated memory: 3240
```

29.4.2. 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.
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.
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**

```cypher
MATCH (KevinB:Person {name: 'Kevin Bacon'}),
    (Al:Person {name: 'Al Pacino'}),
p = shortestPath((KevinB)-[*]-(Al))
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**

```
<table>
<thead>
<tr>
<th>Operator</th>
<th>Details</th>
<th>Estimated Rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>p</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>+Filter</td>
<td>length(p) &gt; $autoint_2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>+ShortestPath</td>
<td>p = (KevinB)-<a href="Al">anon_0*-</a></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>+MultiNodeIndexSeek</td>
<td>BTREE INDEX KevinB:Person(name) WHERE name = $autostring_0,</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Total database accesses: 5, total allocated memory: 3240
```
Chapter 30. Deprecations, additions and compatibility

Cypher is a language that is constantly evolving. New features are added to the language continuously, and occasionally, some features become deprecated and are subsequently removed.

This section lists all of the features that have been removed, deprecated, added, or extended in different Cypher versions. Replacement syntax for deprecated and removed features are also indicated.

30.1. Version 4.4

30.1.1. Deprecated features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATCH (n) RETURN n.propertyName_1, n.propertyName_2 + count(*)</td>
<td>Implied grouping keys are deprecated. Only expressions that do not contain aggregations are still considered grouping keys. In expressions that contain aggregations, the leaves must be either:</td>
</tr>
<tr>
<td></td>
<td>• An aggregation</td>
</tr>
<tr>
<td></td>
<td>• A literal</td>
</tr>
<tr>
<td></td>
<td>• A parameter</td>
</tr>
<tr>
<td></td>
<td>• A variable, ONLY IF it is either:</td>
</tr>
<tr>
<td></td>
<td>1) A projection expression on its own (e.g. the n in RETURN n AS myNode, n.value + count(*)</td>
</tr>
<tr>
<td></td>
<td>2) A local variable in the expression (e.g. the x in RETURN n, n.prop + size([ x IN range(1, 10)</td>
</tr>
<tr>
<td></td>
<td>• Property access, ONLY IF it is also a projection expression on its own (e.g. the n.prop in RETURN n.prop, n.prop + count(*))</td>
</tr>
<tr>
<td></td>
<td>• Map access, ONLY IF it is also a projection expression on its own (e.g. the map.prop in WITH {prop: 2} AS map RETURN map.prop, map.prop + count(*))</td>
</tr>
<tr>
<td>Feature</td>
<td>Details</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Syntax</td>
<td>Replaced by:</td>
</tr>
<tr>
<td>USING PERIODIC COMMIT ...</td>
<td><code>CALL (...) IN TRANSACTIONS</code></td>
</tr>
<tr>
<td>Syntax</td>
<td>CREATE clauses in which a variable introduced in the pattern is also referenced from the same pattern are deprecated.</td>
</tr>
<tr>
<td>CREATE (a {prop:7})-[r:R]-(b {prop: a.prop})</td>
<td>Replaced by: <code>CREATE CONSTRAINT FOR ... REQUIRE ...</code></td>
</tr>
<tr>
<td>Syntax</td>
<td>B-tree indexes are deprecated, partially replaced for now, and will be fully replaced in 5.0 by future indexes. In 4.4, b-tree indexes are still the correct alternative to use.</td>
</tr>
<tr>
<td>CREATE CONSTRAINT ON ... ASSERT ...</td>
<td>B-tree indexes used for string queries are replaced by:</td>
</tr>
<tr>
<td></td>
<td><code>CREATE TEXT INDEX ...</code></td>
</tr>
<tr>
<td></td>
<td>B-tree indexes used for spatial queries will be replaced by:</td>
</tr>
<tr>
<td></td>
<td><code>CREATE POINT INDEX ...</code></td>
</tr>
<tr>
<td></td>
<td>B-tree indexes used for general queries or property value types will be replaced by:</td>
</tr>
<tr>
<td></td>
<td><code>CREATE RANGE INDEX ...</code></td>
</tr>
<tr>
<td></td>
<td>These new indexes may be combined for multiple use cases.</td>
</tr>
<tr>
<td>Feature</td>
<td>Details</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td><strong>Deprecated</strong></td>
</tr>
<tr>
<td>SHOW BTREE INDEXES</td>
<td>B-tree indexes are deprecated, partially replaced for now, and will be fully replaced in 5.0 by future indexes. In 4.4, b-tree indexes are still the correct alternative to use. Replaced by the new and future index types:</td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td><strong>Deprecated</strong></td>
</tr>
<tr>
<td>USING BTREE INDEX</td>
<td>B-tree indexes are deprecated. Replaced by:</td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td><strong>Deprecated</strong></td>
</tr>
<tr>
<td>CREATE CONSTRAINT ... OPTIONS &quot;{&quot; btree-option: btree-value[, ...] &quot;}&quot;</td>
<td>Node key and uniqueness constraints with b-tree options are deprecated and will be replaced in 5.0 by range options, see range indexes. In 4.4, the b-tree index-backed constraints are still the correct alternative to use. Will be replaced by:</td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td><strong>Deprecated</strong></td>
</tr>
<tr>
<td>distance(n.prop, point({x:0, y:0}))</td>
<td>Constraints used for string properties will also require an additional text index to cover the string queries properly. Constraints used for point properties will also require an additional point index to cover the spatial queries properly, see point indexes.</td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td><strong>Deprecated</strong></td>
</tr>
<tr>
<td>point({x:0, y:0}) &lt;= point({x:1, y:1}) &lt;= point({x:2, y:2})</td>
<td>The ability to use the inequality operators &lt;, &lt;=, &gt;, and &gt;= on spatial points is deprecated. Instead, use:</td>
</tr>
</tbody>
</table>
Currently, if n.prop is null, 'one' would be returned. Since null = null returns false in Cypher, a WHEN expression will no longer match in future versions.

Please use IS NULL instead:

```
MATCH (n)
RETURN
CASE
  WHEN n.prop IS NULL THEN 'one'
  ELSE 'two'
END
```

### 30.1.2. New features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Functionality</strong></td>
<td><strong>New</strong></td>
</tr>
<tr>
<td><strong>Deprecated</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Feature</strong></td>
<td><strong>Details</strong></td>
</tr>
<tr>
<td>MATCH (n) RETURN CASE n.prop WHEN null THEN 'one' ELSE 'two' END</td>
<td>Currently, if n.prop is null, 'one' would be returned. Since null = null returns false in Cypher, a WHEN expression will no longer match in future versions. Please use IS NULL instead:</td>
</tr>
<tr>
<td>CALL { ... } IN TRANSACTIONS</td>
<td>New clause for evaluating a subquery in separate transactions. Typically used when modifying or importing large amounts of data. See CALL { ... } IN TRANSACTIONS.</td>
</tr>
<tr>
<td>CREATE CONSTRAINT FOR ... REQUIRE ...</td>
<td>New syntax for creating constraints, applicable to all constraint types.</td>
</tr>
<tr>
<td>CREATE CONSTRAINT [constraint_name] [IF NOT EXISTS] FOR (n:LabelName) REQUIRE (n.propertyName_1, ..., n.propertyName_n) IS UNIQUE [OPTIONS &quot;{&quot; option: value[,...] &quot;}&quot;]</td>
<td>Unique property constraints now allow multiple properties, ensuring that the combination of property values are unique.</td>
</tr>
<tr>
<td>DROP CONSTRAINT ON (n:LabelName) ASSERT (n.propertyName_1, ..., n.propertyName_n) IS UNIQUE</td>
<td>Unique property constraints now allow multiple properties. Replaced by:</td>
</tr>
<tr>
<td></td>
<td>DROP CONSTRAINT name [IF EXISTS]</td>
</tr>
<tr>
<td>Feature</td>
<td>Details</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>Syntax</td>
<td>New</td>
</tr>
<tr>
<td>Functionality</td>
<td>New</td>
</tr>
<tr>
<td>Functionality</td>
<td>New</td>
</tr>
<tr>
<td>Functionality</td>
<td>New</td>
</tr>
<tr>
<td>Functionality</td>
<td>New</td>
</tr>
<tr>
<td>Functionality</td>
<td>New</td>
</tr>
<tr>
<td>Syntax</td>
<td>New</td>
</tr>
<tr>
<td>New privilege:</td>
<td></td>
</tr>
<tr>
<td>Feature</td>
<td>Details</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td><strong>New</strong></td>
</tr>
<tr>
<td>List transactions on the current server.</td>
<td>SHOW TRANSACTION[S] [transaction-id[, ...]] [YIELD *</td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td><strong>New</strong></td>
</tr>
<tr>
<td>Terminate transactions on the current server.</td>
<td>TERMINATE TRANSACTION[S] transaction-id[, ...]</td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td><strong>New</strong></td>
</tr>
<tr>
<td>New Cypher command for modifying a database by changing its access mode.</td>
<td>ALTER DATABASE ... [IF EXISTS] SET ACCESS {READ ONLY</td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td><strong>New</strong></td>
</tr>
<tr>
<td>New privilege that allows a user to modify databases.</td>
<td>ALTER DATABASE</td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td><strong>New</strong></td>
</tr>
<tr>
<td>New privilege that allows a user to modify database access mode.</td>
<td>SET DATABASE ACCESS</td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td><strong>New</strong></td>
</tr>
<tr>
<td>New Cypher command for creating an alias for a database name. Remote aliases are only supported from version 4.4.8.</td>
<td>CREATE ALIAS ... [IF NOT EXISTS] FOR DATABASE ...</td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td><strong>New</strong></td>
</tr>
<tr>
<td>New Cypher command for creating or replacing an alias for a database name. Remote aliases are only supported from version 4.4.8.</td>
<td>CREATE OR REPLACE ALIAS ... FOR DATABASE ...</td>
</tr>
<tr>
<td>Feature</td>
<td>Details</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td>New Cypher command for altering an alias. Remote aliases are only supported from version 4.4.8.</td>
</tr>
<tr>
<td></td>
<td><code>ALTER ALIAS ... [IF EXISTS] SET DATABASE ...</code></td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td>New Cypher command for dropping a database alias.</td>
</tr>
<tr>
<td></td>
<td><code>DROP ALIAS ... [IF EXISTS] FOR DATABASE</code></td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td>New Cypher command for listing database aliases. Only supported since version 4.4.8.</td>
</tr>
<tr>
<td></td>
<td><code>SHOW ALIASES FOR DATABASE</code></td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td>New privilege that allows a user to create, modify, delete and list aliases. Only supported since version 4.4.8.</td>
</tr>
<tr>
<td></td>
<td><code>ALIAS MANAGEMENT</code></td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td>New privilege that allows a user to create aliases. Only supported since version 4.4.8.</td>
</tr>
<tr>
<td></td>
<td><code>CREATE ALIAS</code></td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td>New privilege that allows a user to modify aliases. Only supported since version 4.4.8.</td>
</tr>
<tr>
<td></td>
<td><code>ALTER ALIAS</code></td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td>New privilege that allows a user to delete aliases. Only supported since version 4.4.8.</td>
</tr>
<tr>
<td></td>
<td><code>DROP ALIAS</code></td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td>New privilege that allows a user to show aliases. Only supported since version 4.4.8.</td>
</tr>
<tr>
<td></td>
<td><code>SHOW ALIAS</code></td>
</tr>
</tbody>
</table>
### 30.2. Version 4.3

#### 30.2.1. Deprecated features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td><strong>Deprecated</strong></td>
</tr>
<tr>
<td>CREATE CONSTRAINT [name] ON (node:Label) ASSERT exists(node.property)</td>
<td>Replaced by: CREATE CONSTRAINT [name] ON (node:Label) ASSERT node.property IS NOT NULL</td>
</tr>
<tr>
<td>CREATE CONSTRAINT [name] ON ()-[rel:REL]-() ASSERT exists(rel.property)</td>
<td>Replaced by: CREATE CONSTRAINT [name] ON ()-[rel:REL]-() ASSERT rel.property IS NOT NULL</td>
</tr>
<tr>
<td>exists(prop)</td>
<td>prop IS NOT NULL</td>
</tr>
<tr>
<td>NOT exists(prop)</td>
<td>prop IS NULL</td>
</tr>
<tr>
<td><strong>Syntax</strong></td>
<td><strong>Deprecated</strong></td>
</tr>
<tr>
<td>BRIEF [OUTPUT] for SHOW INDEXES and SHOW CONSTRAINTS.</td>
<td>Replaced by default output columns.</td>
</tr>
<tr>
<td>VERBOSE [OUTPUT] for SHOW INDEXES and SHOW CONSTRAINTS.</td>
<td>Replaced by: YIELD *</td>
</tr>
<tr>
<td>SHOW EXISTS CONSTRAINTS</td>
<td>Still allows BRIEF and VERBOSE but not YIELD or WHERE.</td>
</tr>
</tbody>
</table>
### Feature Details

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Deprecated</th>
<th>Details</th>
</tr>
</thead>
</table>
| **SHOW NODE EXISTS CONSTRAINTS** | | Replaced by: **SHOW NODE [PROPERTY] EXIST[ENCE] CONSTRAINTS**  
Still allows **BRIEF** and **VERBOSE** but not **YIELD** or **WHERE**. |
| **SHOW RELATIONSHIP EXISTS CONSTRAINTS** | | Replaced by: **SHOW RELATIONSHIP [PROPERTY] EXIST[ENCE] CONSTRAINTS**  
Still allows **BRIEF** and **VERBOSE** but not **YIELD** or **WHERE**. |
| **Syntax** Deprecated | | Replaced by: **ON HOME DATABASE**  
For privilege commands: **ON DEFAULT DATABASE** |
| **Syntax** Deprecated | | Replaced by: **ON HOME GRAPH**  
For privilege commands: **ON DEFAULT GRAPH** |
| **MATCH (a) RETURN (a)--()** | | Pattern expressions producing lists of paths are deprecated, but they can still be used as existence predicates, for example in **WHERE** clauses. Instead, use a pattern comprehension: **MATCH (a) RETURN [p=(a)--()] | p]** |

### 30.2.2. Updated features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Functionality Updated</strong></td>
<td>Now allows filtering for: <strong>SHOW INDEXES WHERE ...</strong></td>
</tr>
<tr>
<td></td>
<td><strong>SHOW INDEXES</strong></td>
</tr>
<tr>
<td>Feature</td>
<td>Details</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td>Updated</td>
</tr>
<tr>
<td><strong>SHOW CONSTRAINTS WHERE ...</strong></td>
<td>Now allows filtering for:</td>
</tr>
<tr>
<td></td>
<td><strong>SHOW CONSTRAINTS</strong></td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td>Updated</td>
</tr>
<tr>
<td><strong>SHOW INDEXES YIELD ...</strong></td>
<td>Now allows <strong>YIELD</strong>, <strong>WHERE</strong>, and <strong>RETURN</strong> clauses to</td>
</tr>
<tr>
<td></td>
<td><strong>SHOW INDEXES</strong> to change the output.</td>
</tr>
<tr>
<td></td>
<td><strong>WHERE ...]</strong></td>
</tr>
<tr>
<td></td>
<td><strong>RETURN ...]</strong></td>
</tr>
<tr>
<td><strong>Syntax</strong></td>
<td>Updated</td>
</tr>
<tr>
<td><strong>SHOW [PROPERTY] EXIST[ENCE] CONSTRAINTS</strong></td>
<td>New syntax for filtering <strong>SHOW CONSTRAINTS</strong> on property existence</td>
</tr>
<tr>
<td></td>
<td>constraints.</td>
</tr>
<tr>
<td></td>
<td>Allows <strong>YIELD</strong> and <strong>WHERE</strong> but not <strong>BRIEF</strong> or <strong>VERBOSE</strong>.</td>
</tr>
<tr>
<td><strong>Syntax</strong></td>
<td>Updated</td>
</tr>
<tr>
<td><strong>SHOW NODE [PROPERTY] EXIST[ENCE] CONSTRAINTS</strong></td>
<td>New syntax for filtering <strong>SHOW CONSTRAINTS</strong> on node property</td>
</tr>
<tr>
<td></td>
<td>existence constraints.</td>
</tr>
<tr>
<td></td>
<td>Allows <strong>YIELD</strong> and <strong>WHERE</strong> but not <strong>BRIEF</strong> or <strong>VERBOSE</strong>.</td>
</tr>
<tr>
<td><strong>Syntax</strong></td>
<td>Updated</td>
</tr>
<tr>
<td><strong>SHOW REL[ATIONSHIP] [PROPERTY] EXIST[ENCE] CONSTRAINTS</strong></td>
<td>New syntax for filtering <strong>SHOW CONSTRAINTS</strong> on relationship property</td>
</tr>
<tr>
<td></td>
<td>existence constraints.</td>
</tr>
<tr>
<td></td>
<td>Allows <strong>YIELD</strong> and <strong>WHERE</strong> but not <strong>BRIEF</strong> or <strong>VERBOSE</strong>.</td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td>Updated</td>
</tr>
<tr>
<td><strong>SHOW FULLTEXT INDEXES</strong></td>
<td>Now allows easy filtering for <strong>SHOW INDEXES</strong> on fulltext indexes.</td>
</tr>
<tr>
<td></td>
<td>Allows <strong>YIELD</strong> and <strong>WHERE</strong> but not <strong>BRIEF</strong> or <strong>VERBOSE</strong>.</td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td>Updated</td>
</tr>
<tr>
<td><strong>SHOW LOOKUP INDEXES</strong></td>
<td>Now allows easy filtering for <strong>SHOW INDEXES</strong> on token lookup indexes.</td>
</tr>
<tr>
<td></td>
<td>Allows <strong>YIELD</strong> and <strong>WHERE</strong> but not <strong>BRIEF</strong> or <strong>VERBOSE</strong>.</td>
</tr>
</tbody>
</table>

### 30.2.3. New features
<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
</table>
| **Syntax** New | **CREATE DATABASE** ...
  [OPTIONS {...}] |
| Syntax New | New syntax to pass options to `CREATE DATABASE`. This can be used to specify a specific cluster node to seed data from. |
| Syntax New | **CREATE CONSTRAINT** [name]
  ON (node:Label)
  **ASSERT** node.property IS NOT NULL |
| Syntax New | New syntax for creating node property existence constraints. |
| Syntax New | **CREATE CONSTRAINT** [name]
  ON ()-[rel:REL]-()
  **ASSERT** rel.property IS NOT NULL |
| Syntax New | New syntax for creating relationship property existence constraints. |
| Syntax New | **ALTER USER** name IF EXISTS ...
  SET HOME DATABASE ...
  REMOVE HOME DATABASE |
| Syntax New | Makes altering users idempotent. If the specified name does not exists, no error is thrown. |
| Syntax New | **ALTER USER** ...
  SET HOME DATABASE ...
  REMOVE HOME DATABASE |
| Syntax New | Now allows setting home database for user. |
| Syntax New | **ALTER USER** ...
  REMOVE HOME DATABASE |
| Syntax New | Now allows removing home database for user. |
| Syntax New | **CREATE USER** ...
  SET HOME DATABASE ...
  **SHOW HOME DATABASE** |
<p>| Syntax New | <strong>CREATE USER</strong> now allows setting home database for user. |
| Syntax New | New syntax for showing the home database of the current user. |</p>
<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>New privilege:</td>
</tr>
<tr>
<td><strong>SET USER HOME DATABASE</strong></td>
<td>New Cypher command for administering privilege for changing users home database.</td>
</tr>
<tr>
<td>Syntax</td>
<td>For privilege commands:</td>
</tr>
<tr>
<td><strong>ON HOME DATABASE</strong></td>
<td>New syntax for privileges affecting home database.</td>
</tr>
<tr>
<td>Syntax</td>
<td>For privilege commands:</td>
</tr>
<tr>
<td><strong>ON HOME GRAPH</strong></td>
<td>New syntax for privileges affecting home graph.</td>
</tr>
<tr>
<td>Syntax</td>
<td>New</td>
</tr>
<tr>
<td><strong>CREATE FULLTEXT INDEX ...</strong></td>
<td>Allows creating fulltext indexes on nodes or relationships. They can be dropped by using their name.</td>
</tr>
<tr>
<td>Functionality</td>
<td>New</td>
</tr>
<tr>
<td><strong>CREATE INDEX FOR ()-[r:TYPE]-() ...</strong></td>
<td>Allows creating indexes on relationships with a particular relationship type and property combination. They can be dropped by using their name.</td>
</tr>
<tr>
<td>Functionality</td>
<td>New</td>
</tr>
<tr>
<td><strong>CREATE LOOKUP INDEX ...</strong></td>
<td>Create token lookup index for nodes with any labels or relationships with any relationship type. They can be dropped by using their name.</td>
</tr>
<tr>
<td>Functionality</td>
<td>New</td>
</tr>
<tr>
<td><strong>RENAME ROLE</strong></td>
<td>New Cypher command for changing the name of a role.</td>
</tr>
<tr>
<td>Functionality</td>
<td>New</td>
</tr>
<tr>
<td><strong>RENAME USER</strong></td>
<td>New Cypher command for changing the name of a user.</td>
</tr>
</tbody>
</table>
### New Cypher commands for listing procedures.

```
SHOW PROCEDURE[S]
[EXECUTABLE [BY {CURRENT USER | username}]]
[YIELD ...]
[WHERE ...]
[RETURN ...]
```

### New Cypher commands for listing functions.

```
SHOW [ALL | BUILT IN | USER DEFINED] FUNCTION[S]
[EXECUTABLE [BY {CURRENT USER | username}]]
[YIELD ...]
[WHERE ...]
[RETURN ...]
```

## 30.3. Version 4.2

### 30.3.1. Deprecated features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td>Replaced by ( 0 \ldots ).</td>
</tr>
<tr>
<td>( 0 \ldots )</td>
<td></td>
</tr>
<tr>
<td><strong>Syntax</strong></td>
<td>Only ( 0x \ldots ) (lowercase x) is supported.</td>
</tr>
<tr>
<td>( 0x \ldots )</td>
<td></td>
</tr>
<tr>
<td><strong>Procedure</strong></td>
<td>Replaced by <strong>CREATE INDEX</strong> command.</td>
</tr>
<tr>
<td><code>db.createIndex</code></td>
<td></td>
</tr>
<tr>
<td><strong>Procedure</strong></td>
<td>Replaced by:</td>
</tr>
<tr>
<td><code>db.createNodeKey</code></td>
<td></td>
</tr>
<tr>
<td><strong>Procedure</strong></td>
<td>Replaced by:</td>
</tr>
<tr>
<td><code>db.createUniquePropertyConstraint</code></td>
<td></td>
</tr>
<tr>
<td>Feature</td>
<td>Details</td>
</tr>
<tr>
<td>------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>db.indexes</td>
<td>Replaced by:</td>
</tr>
<tr>
<td>db.indexDetails</td>
<td>Replaced by:</td>
</tr>
<tr>
<td>db.constraints</td>
<td>Replaced by:</td>
</tr>
<tr>
<td>db.schemaStatements</td>
<td>Replaced by:</td>
</tr>
<tr>
<td>Syntax</td>
<td>Unaliased expressions are deprecated in subquery RETURN clauses. Replaced by:</td>
</tr>
</tbody>
</table>

30.3.2. Updated features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functionality</td>
<td>Can now handle multiple roles.</td>
</tr>
<tr>
<td>SHOW ROLE name PRIVILEGES</td>
<td>SHOW ROLES n1, n2, ... PRIVILEGES</td>
</tr>
<tr>
<td>Functionality</td>
<td>Can now handle multiple users.</td>
</tr>
<tr>
<td>SHOW USER name PRIVILEGES</td>
<td>SHOW USERS n1, n2, ... PRIVILEGES</td>
</tr>
</tbody>
</table>
### 30.3.3. New features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Functionality</strong> New</td>
<td>Privileges can now be shown as Cypher commands.</td>
</tr>
<tr>
<td>SHOW PRIVILEGES [AS [REVOKE] COMMAND[S]]</td>
<td></td>
</tr>
<tr>
<td><strong>Syntax</strong> New</td>
<td>New optional part of the Cypher commands for database privileges.</td>
</tr>
<tr>
<td>DEFAULT GRAPH</td>
<td></td>
</tr>
<tr>
<td><strong>Syntax</strong> New</td>
<td>Cypher now interprets literals with prefix (0o) as an octal integer literal.</td>
</tr>
<tr>
<td>(0o)...</td>
<td></td>
</tr>
<tr>
<td><strong>Syntax</strong> New</td>
<td>For <code>CREATE USER</code> and <code>ALTER USER</code>, it is now possible to set (or update) a password when the plaintext password is unknown, but the encrypted password is available.</td>
</tr>
<tr>
<td>SET [PLAINTEXT</td>
<td>ENCRYPTED] PASSWORD</td>
</tr>
<tr>
<td><strong>Functionality</strong> New</td>
<td>New Cypher commands for administering privileges for executing procedures and user defined functions. See The DBMS EXECUTE privileges.</td>
</tr>
<tr>
<td>New privilege:</td>
<td>EXECUTE</td>
</tr>
<tr>
<td><strong>Syntax</strong> New</td>
<td>Allows setting index provider and index configuration when creating an index.</td>
</tr>
<tr>
<td>CREATE [BTREE] INDEX ... [OPTIONS [...]</td>
<td></td>
</tr>
<tr>
<td>Feature</td>
<td>Details</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>Syntax New</td>
<td><strong>CREATE CONSTRAINT ... IS NODE KEY [OPTIONS (...)]</strong>&lt;br&gt;Allows setting index provider and index configuration for the backing index when creating a node key constraint.</td>
</tr>
<tr>
<td>Syntax New</td>
<td><strong>CREATE CONSTRAINT ... IS UNIQUE [OPTIONS (...)]</strong>&lt;br&gt;Allows setting index provider and index configuration for the backing index when creating a uniqueness constraint.</td>
</tr>
<tr>
<td>Syntax New</td>
<td><strong>SHOW CURRENT USER</strong>&lt;br&gt;New Cypher command for showing current logged-in user and roles.</td>
</tr>
<tr>
<td>Functionality New</td>
<td>**SHOW [ALL</td>
</tr>
<tr>
<td>Functionality New</td>
<td><strong>New privilege:</strong>&lt;br&gt;<strong>SHOW INDEX</strong>&lt;br&gt;New Cypher command for administering privilege for listing indexes.</td>
</tr>
<tr>
<td>Functionality New</td>
<td><strong>New privilege:</strong>&lt;br&gt;<strong>SHOW CONSTRAINT</strong>&lt;br&gt;New Cypher command for administering privilege for listing constraints.</td>
</tr>
</tbody>
</table>

30.4. Version 4.1.3

30.4.1. New features
<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax New</td>
<td>Makes index creation idempotent. If an index with the name or schema already exists no error will be thrown.</td>
</tr>
<tr>
<td><code>CREATE INDEX [name] IF NOT EXISTS FOR ...</code></td>
<td></td>
</tr>
<tr>
<td>Syntax New</td>
<td>Makes index deletion idempotent. If no index with the name exists no error will be thrown.</td>
</tr>
<tr>
<td><code>DROP INDEX name IF EXISTS</code></td>
<td></td>
</tr>
<tr>
<td>Syntax New</td>
<td>Makes constraint creation idempotent. If a constraint with the name or type and schema already exists no error will be thrown.</td>
</tr>
<tr>
<td><code>CREATE CONSTRAINT [name] IF NOT EXISTS ON ...</code></td>
<td></td>
</tr>
<tr>
<td>Syntax New</td>
<td>Makes constraint deletion idempotent. If no constraint with the name exists no error will be thrown.</td>
</tr>
<tr>
<td><code>DROP CONSTRAINT name IF EXISTS</code></td>
<td></td>
</tr>
</tbody>
</table>

### 30.5. Version 4.1

#### 30.5.1. Restricted features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functionality Restricted</td>
<td>No longer revokes sub-privileges when revoking a compound privilege, e.g. when revoking <code>INDEX MANAGEMENT</code>, any <code>CREATE INDEX</code> and <code>DROP INDEX</code> privileges will no longer be revoked.</td>
</tr>
<tr>
<td><code>REVOKE ...</code></td>
<td></td>
</tr>
<tr>
<td>Functionality Restricted</td>
<td>No longer includes the privileges <code>START DATABASE</code> and <code>STOP DATABASE</code>.</td>
</tr>
<tr>
<td><code>ALL DATABASE PRIVILEGES</code></td>
<td></td>
</tr>
</tbody>
</table>

#### 30.5.2. Updated features
### 30.5.3. New features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
</table>
| **Procedure** | The queryId procedure format has changed, and no longer includes the database name. For example, mydb-query-123 is now query-123. This change affects built-in procedures `dbms.listQueries()`, `dbms.listActiveLocks(queryId)`, `dbms.killQueries(queryIds)` and `dbms.killQuery(queryId)`.
| queryId       |                                                                                                                                       |
| **Functionality** | The returned privileges are a closer match to the original grants and denies, e.g. if granted MATCH the command will show that specific privilege and not the TRAVERSE and READ privileges. Added support for YIELD and WHERE clauses to allow filtering results. |
| SHOW PRIVILEGES |                                                                                                                                       |

#### New role:
- **PUBLIC**

The PUBLIC role is automatically assigned to all users, giving them a set of base privileges.

#### Syntax:
- **MATCH**

The MATCH privilege can now be revoked.

#### Functionality:
- **SHOW USERS**

New support for YIELD and WHERE clauses to allow filtering results.

- **SHOW ROLES**

New support for YIELD and WHERE clauses to allow filtering results.

- **SHOW DATABASES**

New support for YIELD and WHERE clauses to allow filtering results.
<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Functionality</strong></td>
<td><strong>New</strong></td>
</tr>
<tr>
<td><strong>TRANSACTION MANAGEMENT</strong> privileges</td>
<td>New Cypher commands for administering transaction management.</td>
</tr>
<tr>
<td><strong>DBMS USER MANAGEMENT</strong> privileges</td>
<td>New Cypher commands for administering user management.</td>
</tr>
<tr>
<td><strong>DBMS DATABASE MANAGEMENT</strong> privileges</td>
<td>New Cypher commands for administering database management.</td>
</tr>
<tr>
<td><strong>DBMS PRIVILEGE MANAGEMENT</strong> privileges</td>
<td>New Cypher commands for administering privilege management.</td>
</tr>
<tr>
<td><strong>ALL DBMS PRIVILEGES</strong></td>
<td>New Cypher command for administering role, user, database and privilege management.</td>
</tr>
<tr>
<td><strong>ALL GRAPH PRIVILEGES</strong></td>
<td>New Cypher command for administering read and write privileges.</td>
</tr>
<tr>
<td><strong>Write privileges</strong></td>
<td>New Cypher commands for administering write privileges.</td>
</tr>
<tr>
<td><strong>ON DEFAULT DATABASE</strong></td>
<td>New optional part of the Cypher commands for database privileges.</td>
</tr>
</tbody>
</table>

30.6. Version 4.0

30.6.1. Removed features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function</strong></td>
<td><strong>Removed</strong></td>
</tr>
<tr>
<td><strong>rels()</strong></td>
<td>Replaced by relationships().</td>
</tr>
<tr>
<td>Feature</td>
<td>Details</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td><strong>Function Removed</strong></td>
<td>Replaced by <code>toInt()</code>.</td>
</tr>
<tr>
<td><code>toInt()</code></td>
<td></td>
</tr>
<tr>
<td><strong>Function Removed</strong></td>
<td>Replaced by <code>toLowerCase()</code>.</td>
</tr>
<tr>
<td><code>toLowerCase()</code></td>
<td></td>
</tr>
<tr>
<td><strong>Function Removed</strong></td>
<td>Replaced by <code>toUpperCase()</code>.</td>
</tr>
<tr>
<td><code>toUpperCase()</code></td>
<td></td>
</tr>
<tr>
<td><strong>Function Removed</strong></td>
<td>Replaced by list comprehension.</td>
</tr>
<tr>
<td><code>extract()</code></td>
<td></td>
</tr>
<tr>
<td><strong>Function Removed</strong></td>
<td>Replaced by list comprehension.</td>
</tr>
<tr>
<td><code>filter()</code></td>
<td></td>
</tr>
<tr>
<td><strong>Functionality Removed</strong></td>
<td>The <code>RULE</code> planner was removed in 3.2, but still possible to trigger using <code>START</code> or <code>CREATE UNIQUE</code> clauses. Now it is completely removed.</td>
</tr>
<tr>
<td>For Rule planner:</td>
<td><code>CYPHER planner=rule</code></td>
</tr>
<tr>
<td><strong>Functionality Removed</strong></td>
<td>The removal of the <code>RULE</code> planner in 3.2 was the beginning of the end for explicit indexes. Now they are completely removed, including the removal of the built-in procedures for Neo4j 3.3 to 3.5.</td>
</tr>
<tr>
<td>Explicit indexes</td>
<td></td>
</tr>
<tr>
<td><strong>Functionality Removed</strong></td>
<td>Replaced by the new pipelined runtime which covers a much wider range of queries.</td>
</tr>
<tr>
<td>For compiled runtime:</td>
<td><code>CYPHER runtime=compiled</code></td>
</tr>
<tr>
<td><strong>Clause Removed</strong></td>
<td>Running queries with this clause will cause a syntax error. Running with <code>CYPHER 3.5</code> will cause a runtime error due to the removal of the rule planner.</td>
</tr>
<tr>
<td><code>CREATE UNIQUE</code></td>
<td></td>
</tr>
</tbody>
</table>
### 30.6.2. Deprecated features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Clause</strong> Removed</td>
<td>Running queries with this clause will cause a syntax error. Running with CYpher 3.5 will cause a runtime error due to the removal of the rule planner.</td>
</tr>
<tr>
<td><strong>Syntax Removed</strong></td>
<td>Replaced by `MATCH (n)-[:A</td>
</tr>
<tr>
<td><strong>Syntax Removed</strong></td>
<td>Replaced by `MATCH (n)-[x:A</td>
</tr>
<tr>
<td><strong>Syntax Removed</strong></td>
<td>Replaced by `MATCH (n)-[x:A</td>
</tr>
<tr>
<td><strong>Syntax Removed</strong></td>
<td>Replaced by <code>$parameter</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax Deprecated</strong></td>
<td>As in Cypher 3.2, this is replaced by:</td>
</tr>
<tr>
<td><strong>Syntax Deprecated</strong></td>
<td>Replaced by <code>CREATE INDEX FOR (n:Label) ON (n.prop)</code></td>
</tr>
<tr>
<td><strong>Syntax Deprecated</strong></td>
<td>Replaced by <code>DROP INDEX name</code></td>
</tr>
</tbody>
</table>
### 30.6.3. Restricted features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function</strong></td>
<td>Restricted</td>
</tr>
<tr>
<td><code>length()</code></td>
<td>Restricted to only work on paths. See <code>length()</code> for more details.</td>
</tr>
<tr>
<td><strong>Function</strong></td>
<td>Restricted</td>
</tr>
<tr>
<td><code>size()</code></td>
<td>No longer works for paths. Only works for strings, lists and pattern expressions. See <code>size()</code> for more details.</td>
</tr>
</tbody>
</table>

### 30.6.4. Updated features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td>Extended</td>
</tr>
<tr>
<td><code>CREATE CONSTRAINT [name] ON ...</code></td>
<td>The create constraint syntax can now include a name.</td>
</tr>
</tbody>
</table>
### 30.6.5. New features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Functionality</strong> New</td>
<td>This Neo4j Enterprise Edition only feature involves a new runtime that has many performance enhancements.</td>
</tr>
<tr>
<td>Pipelined runtime:</td>
<td></td>
</tr>
<tr>
<td>CYPHER runtime=pipelined</td>
<td></td>
</tr>
<tr>
<td><strong>Functionality</strong> New</td>
<td>New Cypher commands for administering multiple databases.</td>
</tr>
<tr>
<td>Multi-database administration</td>
<td></td>
</tr>
<tr>
<td><strong>Functionality</strong> New</td>
<td>New Cypher commands for administering role-based access control.</td>
</tr>
<tr>
<td>Access control</td>
<td></td>
</tr>
<tr>
<td><strong>Functionality</strong> New</td>
<td>New Cypher commands for administering dbms, database, graph and sub-graph access control.</td>
</tr>
<tr>
<td>Fine-grained security</td>
<td></td>
</tr>
<tr>
<td><strong>Syntax</strong> New</td>
<td>New syntax for creating indexes, which can include a name.</td>
</tr>
<tr>
<td>CREATE INDEX [name] FOR (n:Label) ON (n.prop)</td>
<td></td>
</tr>
<tr>
<td><strong>Syntax</strong> New</td>
<td>New command for dropping an index by name.</td>
</tr>
<tr>
<td>DROP INDEX name</td>
<td></td>
</tr>
<tr>
<td><strong>Syntax</strong> New</td>
<td>New command for dropping a constraint by name, no matter the type.</td>
</tr>
<tr>
<td>DROP CONSTRAINT name</td>
<td></td>
</tr>
<tr>
<td><strong>Clause</strong> New</td>
<td>Existential sub-queries are sub-clauses used to filter the results of a MATCH, OPTIONAL MATCH, or WITH clause.</td>
</tr>
<tr>
<td>WHERE EXISTS {...}</td>
<td></td>
</tr>
<tr>
<td><strong>Clause</strong> New</td>
<td>New clause to specify which graph a query, or query part, is executed against.</td>
</tr>
<tr>
<td>USE neo4j</td>
<td></td>
</tr>
</tbody>
</table>
30.7. Version 3.5

30.7.1. Deprecated features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Functionality</strong></td>
<td><strong>Deprecated</strong></td>
</tr>
<tr>
<td>Compiled runtime:</td>
<td>The compiled runtime will be discontinued in the next major release. It might still be used for default queries in order to not cause regressions, but explicitly requesting it will not be possible.</td>
</tr>
<tr>
<td><code>CYPER runtime=compiled</code></td>
<td></td>
</tr>
<tr>
<td><strong>Function</strong></td>
<td><strong>Deprecated</strong></td>
</tr>
<tr>
<td><code>extract()</code></td>
<td>Replaced by list comprehension.</td>
</tr>
<tr>
<td><code>filter()</code></td>
<td>Replaced by list comprehension.</td>
</tr>
</tbody>
</table>

30.8. Version 3.4

<table>
<thead>
<tr>
<th>Feature</th>
<th>Type</th>
<th>Change</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Spatial point types</strong></td>
<td>Functionality</td>
<td>Amendment</td>
<td>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.</td>
</tr>
<tr>
<td><code>point() - Cartesian 3D</code></td>
<td>Function</td>
<td>Added</td>
<td></td>
</tr>
<tr>
<td><code>point() - WGS 84 3D</code></td>
<td>Function</td>
<td>Added</td>
<td></td>
</tr>
<tr>
<td><code>randomUUID()</code></td>
<td>Function</td>
<td>Added</td>
<td></td>
</tr>
<tr>
<td><strong>Temporal types</strong></td>
<td>Functionality</td>
<td>Added</td>
<td>Supports storing, indexing and working with the following temporal types: Date, Time, LocalTime, DateTime, LocalDateTime and Duration.</td>
</tr>
<tr>
<td>Feature</td>
<td>Type</td>
<td>Change</td>
<td>Details</td>
</tr>
<tr>
<td>-------------------------</td>
<td>------------</td>
<td>--------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Temporal functions</td>
<td>Functionality</td>
<td>Added</td>
<td>Functions allowing for the creation and manipulation of values for each temporal type — Date, Time, LocalTime, DateTime, LocalDateTime and Duration.</td>
</tr>
<tr>
<td>Temporal operators</td>
<td>Functionality</td>
<td>Added</td>
<td>Operators allowing for the manipulation of values for each temporal type — Date, Time, LocalTime, DateTime, LocalDateTime and Duration.</td>
</tr>
<tr>
<td>toString()</td>
<td>Function</td>
<td>Extended</td>
<td>Now also allows temporal values as input (i.e. values of type Date, Time, LocalTime, DateTime, LocalDateTime or Duration).</td>
</tr>
</tbody>
</table>

30.9. Version 3.3

<table>
<thead>
<tr>
<th>Feature</th>
<th>Type</th>
<th>Change</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
<td>Clause</td>
<td>Removed</td>
<td>As in Cypher 3.2, any queries using the START clause will revert back to Cypher 3.1 planner=rule. However, there are built-in procedures for Neo4j versions 3.3 to 3.5 for accessing explicit indexes. The procedures will enable users to use the current version of Cypher and the cost planner together with these indexes. An example of this is CALL db.index.explicit.searchNodes('my_index','email:me*').</td>
</tr>
<tr>
<td>CYpher runtime=slotted</td>
<td>Functionality</td>
<td>Added</td>
<td>Neo4j Enterprise Edition only</td>
</tr>
<tr>
<td>(Faster interpreted runtime)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>max(), min()</td>
<td>Function</td>
<td>Extended</td>
<td>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</td>
</tr>
</tbody>
</table>
### 30.10. Version 3.2

<table>
<thead>
<tr>
<th>Feature</th>
<th>Type</th>
<th>Change</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>CYPHER planner=rule (Rule planner)</td>
<td>Functionality</td>
<td>Removed</td>
<td>All queries now use the cost planner. Any query prepended thus will fall back to using Cypher 3.1.</td>
</tr>
<tr>
<td>CREATE UNIQUE Clause</td>
<td>Clause</td>
<td>Removed</td>
<td>Running such queries will fall back to using Cypher 3.1 (and use the rule planner)</td>
</tr>
<tr>
<td>START Clause</td>
<td>Clause</td>
<td>Removed</td>
<td>Running such queries will fall back to using Cypher 3.1 (and use the rule planner)</td>
</tr>
<tr>
<td>MATCH (n)-[rs*]-() RETURN rs</td>
<td>Syntax</td>
<td>Deprecated</td>
<td>Replaced by MATCH p=(n)-[*]-() RETUN relationships(p) AS rs</td>
</tr>
<tr>
<td>MATCH (n)-[:A</td>
<td>B]:C {foo: 'bar'}-() RETURN n</td>
<td>Syntax</td>
<td>Deprecated</td>
</tr>
<tr>
<td>MATCH (n)-[x:A</td>
<td>B]:C]-() RETURN n</td>
<td>Syntax</td>
<td>Deprecated</td>
</tr>
<tr>
<td>MATCH (n)-[x:A</td>
<td>B</td>
<td>C*]-() RETURN n</td>
<td>Syntax</td>
</tr>
<tr>
<td>User-defined aggregation functions</td>
<td>Functionality</td>
<td>Added</td>
<td></td>
</tr>
<tr>
<td>Composite indexes</td>
<td>Index</td>
<td>Added</td>
<td></td>
</tr>
<tr>
<td>Node Key</td>
<td>Index</td>
<td>Added</td>
<td>Neo4j Enterprise Edition only</td>
</tr>
<tr>
<td>CYPHER runtime=compiled (Compiled runtime)</td>
<td>Functionality</td>
<td>Added</td>
<td>Neo4j Enterprise Edition only</td>
</tr>
<tr>
<td>reverse()</td>
<td>Function</td>
<td>Extended</td>
<td>Now also allows a list as input</td>
</tr>
<tr>
<td>max(), min()</td>
<td>Function</td>
<td>Extended</td>
<td>Now also supports aggregation over a set containing both strings and numbers</td>
</tr>
</tbody>
</table>

### 30.11. Version 3.1

<table>
<thead>
<tr>
<th>Feature</th>
<th>Type</th>
<th>Change</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>rels()</td>
<td>Function</td>
<td>Deprecated</td>
<td>Replaced by relationships()</td>
</tr>
<tr>
<td>toInt()</td>
<td>Function</td>
<td>Deprecated</td>
<td>Replaced by toInteger()</td>
</tr>
<tr>
<td>lower()</td>
<td>Function</td>
<td>Deprecated</td>
<td>Replaced by toLower()</td>
</tr>
<tr>
<td>upper()</td>
<td>Function</td>
<td>Deprecated</td>
<td>Replaced by toUpper()</td>
</tr>
</tbody>
</table>
30.12. Version 3.0

<table>
<thead>
<tr>
<th>Feature</th>
<th>Type</th>
<th>Change</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>has()</td>
<td>Function</td>
<td>Removed</td>
<td>Replaced by exists()</td>
</tr>
<tr>
<td>str()</td>
<td>Function</td>
<td>Removed</td>
<td>Replaced by toString()</td>
</tr>
<tr>
<td>{parameter}</td>
<td>Syntax</td>
<td>Deprecated</td>
<td>Replaced by $parameter</td>
</tr>
<tr>
<td>properties()</td>
<td>Function</td>
<td>Added</td>
<td></td>
</tr>
<tr>
<td>CALL [..YIELD]</td>
<td>Clause</td>
<td>Added</td>
<td></td>
</tr>
<tr>
<td>point() - Cartesian 2D</td>
<td>Function</td>
<td>Added</td>
<td></td>
</tr>
<tr>
<td>point() - WGS 84 2D</td>
<td>Function</td>
<td>Added</td>
<td></td>
</tr>
<tr>
<td>distance()</td>
<td>Function</td>
<td>Added</td>
<td></td>
</tr>
<tr>
<td>User-defined procedures</td>
<td>Functionality</td>
<td>Added</td>
<td></td>
</tr>
<tr>
<td>toString()</td>
<td>Function</td>
<td>Extended</td>
<td>Now also allows Boolean values as input</td>
</tr>
</tbody>
</table>

30.13. Compatibility

Neo4j's ability to support multiple older versions of the Cypher language has changed over time. In versions prior to Neo4j 3.4, the backwards compatibility layer included the Cypher language parser, planner, and runtime. All supported versions of Cypher ran on the same Neo4j kernel. However, this changed in Neo4j 3.4 when the runtime was excluded from the compatibility layer. When you run, e.g. a `CYPHER 3.1` query in Neo4j 3.5, the query is planned with the 3.1 planner, but run with 3.5 runtime and kernel. The compatibility layer changed again in Neo4j 4.0 and it now includes only the Cypher language parser. When you run a `CYPHER 3.5` query, e.g., in Neo4j 4.4, Neo4j parses the older language features, but uses the 4.4 planner, runtime, and kernel to plan and run the query. The primary reason for these changes is the optimizations in the Cypher runtime to allow Cypher queries to perform better.

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 3.5` will execute the query with the version of Cypher included in Neo4j 3.5.

Below is an example using the older parameter syntax `{param}`:

```cypher
CYPHER 3.5
MATCH (n:Person)
WHERE n.age > {agelimit}
RETURN n.name, n.age
```

Without the `CYPHER 3.5` prefix this query would fail with a syntax error. With `CYPHER 3.5` however, it will only generate a warning and still work.

30.14. Supported language versions

Neo4j 4.4 supports the following versions of the Cypher language:

- Neo4j Cypher 3.5
- Neo4j Cypher 4.3
- Neo4j Cypher 4.4

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 31. 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
- Administrative commands
- Privilege Actions

31.1. Clauses

<table>
<thead>
<tr>
<th>Clause</th>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL [... YIELD]</td>
<td>Reading/Writing</td>
<td>Invoke a procedure deployed in the database.</td>
</tr>
<tr>
<td>CALL {...}</td>
<td>Reading/Writing</td>
<td>Evaluates a subquery, typically used for post-union processing or aggregations</td>
</tr>
<tr>
<td>CREATE</td>
<td>Writing</td>
<td>Create nodes and relationships.</td>
</tr>
<tr>
<td>CREATE CONSTRAINT [existence]</td>
<td>Schema</td>
<td>Create a constraint ensuring that all nodes with a particular label have a certain property.</td>
</tr>
<tr>
<td>CREATE CONSTRAINT [node_key]</td>
<td>Schema</td>
<td>Create a constraint that ensures 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.</td>
</tr>
<tr>
<td>CREATE CONSTRAINT [existence]</td>
<td>Schema</td>
<td>Create a constraint that ensures all relationships with a particular type have a certain property.</td>
</tr>
<tr>
<td>CREATE CONSTRAINT [uniqueness]</td>
<td>Schema</td>
<td>Create a constraint that ensures the uniqueness of the combination of node label and property values for a particular property key combination across all nodes.</td>
</tr>
<tr>
<td>Clause</td>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>CREATE [BTREE] INDEX [single] [IF NOT EXISTS] FOR (n:Label) ON (n.property) [OPTIONS {optionKey: optionValue[, ...]}]</td>
<td>Schema</td>
<td>Create an index on all nodes with a particular label and a single property; i.e. create a single-property index.</td>
</tr>
<tr>
<td>CREATE [BTREE] INDEX [single] [IF NOT EXISTS] FOR ()-[&quot;r:TYPE&quot;]-() ON (r.property) [OPTIONS {optionKey: optionValue[, ...]}]</td>
<td>Schema</td>
<td>Create an index on all relationships with a particular relationship type and a single property; i.e. create a single-property index.</td>
</tr>
<tr>
<td>CREATE [BTREE] INDEX [composite] [IF NOT EXISTS] FOR (n:Label) ON (n.prop1, ..., n.propN) [OPTIONS {optionKey: optionValue[, ...]}]</td>
<td>Schema</td>
<td>Create an index on all nodes with a particular label and multiple properties; i.e. create a composite index.</td>
</tr>
<tr>
<td>CREATE [BTREE] INDEX [composite] [IF NOT EXISTS] FOR ()-[&quot;r:TYPE&quot;]-() ON (r.prop1, ..., r.propN) [OPTIONS {optionKey: optionValue[, ...]}]</td>
<td>Schema</td>
<td>Create an index on all relationships with a particular relationship type and multiple properties; i.e. create a composite index.</td>
</tr>
<tr>
<td>CREATE FULLTEXT INDEX [name] [IF NOT EXISTS] FOR (n:Label[&quot;</td>
<td>&quot; ... &quot;</td>
<td>&quot; LabelN]) ON EACH &quot;[&quot; n.property[&quot;, ... n.propertyN] &quot;&quot;] [OPTIONS {optionKey: optionValue[, ...]}]</td>
</tr>
<tr>
<td>CREATE FULLTEXT INDEX [name] [IF NOT EXISTS] FOR ()-[&quot;r:TYPE&quot;]-[&quot;</td>
<td>&quot; ... &quot;</td>
<td>&quot; TYPE_N] ON EACH &quot;[&quot; r.property[, ... r.propertyN] &quot;&quot;] [OPTIONS {optionKey: optionValue[, ...]}]</td>
</tr>
<tr>
<td>CREATE LOOKUP INDEX [name] [IF NOT EXISTS] FOR (n) ON EACH labels(n) [OPTIONS {optionKey: optionValue[, ...]}]</td>
<td>Schema</td>
<td>Create an index on all nodes with any label.</td>
</tr>
<tr>
<td>CREATE LOOKUP INDEX [name] [IF NOT EXISTS] FOR ()-[&quot;r&quot;]-() ON [EACH] type(r) [OPTIONS {optionKey: optionValue[, ...]}]</td>
<td>Schema</td>
<td>Create an index on all relationships with any relationship type.</td>
</tr>
<tr>
<td>CREATE TEXT INDEX [name] [IF NOT EXISTS] FOR (n:Label) ON (n.property) [OPTIONS {optionKey: optionValue[, ...]}]</td>
<td>Schema</td>
<td>Create a text index on nodes.</td>
</tr>
<tr>
<td>CREATE TEXT INDEX [name] [IF NOT EXISTS] FOR ()-[&quot;r:TYPE&quot;]-() ON (r.property) [OPTIONS {optionKey: optionValue[, ...]}]</td>
<td>Schema</td>
<td>Create a text index on relationships.</td>
</tr>
<tr>
<td>DELETE</td>
<td>Writing</td>
<td>Delete nodes, relationships or paths. Any node to be deleted must also have all associated relationships explicitly deleted.</td>
</tr>
<tr>
<td>Clause</td>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DETACH DELETE</td>
<td>Writing</td>
<td>Delete a node or set of nodes. All associated relationships will automatically be deleted.</td>
</tr>
<tr>
<td>DROP CONSTRAINT name [IF EXISTS]</td>
<td>Schema</td>
<td>Drop a constraint using the name.</td>
</tr>
<tr>
<td>DROP INDEX name [IF EXISTS]</td>
<td>Schema</td>
<td>Drop an index using the name.</td>
</tr>
<tr>
<td>FOREACH</td>
<td>Writing</td>
<td>Update data within a list, whether components of a path, or the result of aggregation.</td>
</tr>
<tr>
<td>LIMIT</td>
<td>Reading sub-clause</td>
<td>A sub-clause used to constrain the number of rows in the output.</td>
</tr>
<tr>
<td>LOAD CSV</td>
<td>Importing data</td>
<td>Use when importing data from CSV files.</td>
</tr>
<tr>
<td>MATCH</td>
<td>Reading</td>
<td>Specify the patterns to search for in the database.</td>
</tr>
<tr>
<td>MERGE</td>
<td>Reading/Writing</td>
<td>Ensures that a pattern exists in the graph. Either the pattern already exists, or it needs to be created.</td>
</tr>
<tr>
<td>ON CREATE</td>
<td>Reading/Writing</td>
<td>Used in conjunction with MERGE, specifying the actions to take if the pattern needs to be created.</td>
</tr>
<tr>
<td>ON MATCH</td>
<td>Reading/Writing</td>
<td>Used in conjunction with MERGE, specifying the actions to take if the pattern already exists.</td>
</tr>
<tr>
<td>OPTIONAL MATCH</td>
<td>Reading</td>
<td>Specify the patterns to search for in the database while using nulls for missing parts of the pattern.</td>
</tr>
<tr>
<td>ORDER BY [ASC</td>
<td>ENDING]</td>
<td>DESC[ENDING]]</td>
</tr>
<tr>
<td>REMOVE</td>
<td>Writing</td>
<td>Remove properties and labels from nodes and relationships.</td>
</tr>
<tr>
<td>RETURN ... [AS]</td>
<td>Projecting</td>
<td>Defines what to include in the query result set.</td>
</tr>
<tr>
<td>SET</td>
<td>Writing</td>
<td>Update labels on nodes and properties on nodes and relationships.</td>
</tr>
<tr>
<td>SHOW [ALL</td>
<td>UNIQUE</td>
<td>NODE [PROPERTY \</td>
</tr>
<tr>
<td>ENCE][RE</td>
<td>ATIONSHIP]</td>
<td></td>
</tr>
<tr>
<td>[PROPERTY] EXIST[ENCE]][PROPERTY]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXIST[ENCE]][PROPERTY] EXIST[ENCE]]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NODE KEY] CONSTRAINT[S]]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clause</td>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>**SHOW [ALL</td>
<td>BTREE</td>
<td>FULLTEXT</td>
</tr>
<tr>
<td>**SHOW [ALL</td>
<td>BUILT IN</td>
<td>USER DEFINED</td>
</tr>
<tr>
<td>**SHOW PROCEDURE[S]\ [EXECUTABLE \ [BY \ [CURRENT USER]username]</td>
<td>]]**</td>
<td><strong>DBMS</strong></td>
</tr>
<tr>
<td>**SHOW TRANSACTION[S]\ transaction-id[, ...]</td>
<td>]**</td>
<td><strong>DBMS</strong></td>
</tr>
<tr>
<td><strong>SKIP</strong></td>
<td><strong>Reading/Writing</strong></td>
<td>A sub-clause defining from which row to start including the rows in the output.</td>
</tr>
<tr>
<td>**TERMINATE TRANSACTION[S]\ transaction-id[, ...]</td>
<td>]**</td>
<td><strong>DBMS</strong></td>
</tr>
<tr>
<td><strong>UNION</strong></td>
<td><strong>Set operations</strong></td>
<td>Combines the result of multiple queries. Duplicates are removed.</td>
</tr>
<tr>
<td><strong>UNION ALL</strong></td>
<td><strong>Set operations</strong></td>
<td>Combines the result of multiple queries. Duplicates are retained.</td>
</tr>
<tr>
<td><strong>UNWIND ... [AS]</strong></td>
<td><strong>Projecting</strong></td>
<td>Expands a list into a sequence of rows.</td>
</tr>
<tr>
<td><strong>USE</strong></td>
<td><strong>Multiple graphs</strong></td>
<td>Determines which graph a query, or query part, is executed against.</td>
</tr>
<tr>
<td><strong>USING INDEX variable:Label(property)</strong></td>
<td><strong>Hint</strong></td>
<td>Index hints are used to specify which index, if any, the planner should use as a starting point.</td>
</tr>
<tr>
<td><strong>USING INDEX SEEK variable:Label(property)</strong></td>
<td><strong>Hint</strong></td>
<td>Index seek hint instructs the planner to use an index seek for this clause.</td>
</tr>
<tr>
<td><strong>USING JOIN ON variable</strong></td>
<td><strong>Hint</strong></td>
<td>Join hints are used to enforce a join operation at specified points.</td>
</tr>
<tr>
<td><strong>USING PERIODIC COMMIT</strong></td>
<td><strong>Hint</strong></td>
<td>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>.</td>
</tr>
<tr>
<td><strong>USING SCAN variable:Label</strong></td>
<td><strong>Hint</strong></td>
<td>Scan hints are used to force the planner to do a label scan (followed by a filtering operation) instead of using an index.</td>
</tr>
<tr>
<td><strong>WITH ... [AS]</strong></td>
<td><strong>Projecting</strong></td>
<td>Allows query parts to be chained together, piping the results from one to be used as starting points or criteria in the next.</td>
</tr>
<tr>
<td>Clause</td>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>WHERE</strong></td>
<td>Reading sub-clause</td>
<td>A sub-clause used to add constraints to the patterns in a MATCH or OPTIONAL MATCH clause, or to filter the results of a WITH clause.</td>
</tr>
<tr>
<td><strong>WHERE EXISTS {..}</strong></td>
<td>Reading sub-clause</td>
<td>An existential sub-query used to filter the results of a MATCH, OPTIONAL MATCH or WITH clause.</td>
</tr>
</tbody>
</table>

31.2. Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>Mathematical</td>
<td>Modulo division</td>
</tr>
<tr>
<td>*</td>
<td>Mathematical</td>
<td>Multiplication</td>
</tr>
<tr>
<td>*</td>
<td>Temporal</td>
<td>Multiplying a duration with a number</td>
</tr>
<tr>
<td>+</td>
<td>Mathematical</td>
<td>Addition</td>
</tr>
<tr>
<td>+</td>
<td>String</td>
<td>Concatenation</td>
</tr>
<tr>
<td>&lt;&lt;query-operators-property, +⇒&gt;</td>
<td>Property</td>
<td>Property mutation</td>
</tr>
<tr>
<td>+</td>
<td>List</td>
<td>Concatenation</td>
</tr>
<tr>
<td>+</td>
<td>Temporal</td>
<td>Adding two durations, or a duration and a temporal instant</td>
</tr>
<tr>
<td>&lt;&lt;query-operators-mathematical, →&gt;</td>
<td>Mathematical</td>
<td>Subtraction or unary minus</td>
</tr>
<tr>
<td>&lt;&lt;query-operators-temporal, +⇒&gt;</td>
<td>Temporal</td>
<td>Subtracting a duration from a temporal instant or from another duration</td>
</tr>
<tr>
<td>.</td>
<td>Map</td>
<td>Static value access by key</td>
</tr>
<tr>
<td>.</td>
<td>Property</td>
<td>Static property access</td>
</tr>
<tr>
<td>/</td>
<td>Mathematical</td>
<td>Division</td>
</tr>
<tr>
<td>/</td>
<td>Temporal</td>
<td>Dividing a duration by a number</td>
</tr>
<tr>
<td>&lt;</td>
<td>Comparison</td>
<td>Less than</td>
</tr>
<tr>
<td>&lt;&lt;query-operators-comparison, &lt;⇒&gt;</td>
<td>Comparison</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Comparison</td>
<td>Inequality</td>
</tr>
<tr>
<td>&lt;&lt;query-operators-comparison, →⇒&gt;</td>
<td>Comparison</td>
<td>Equality</td>
</tr>
<tr>
<td>&lt;&lt;query-operators-property, →⇒&gt;</td>
<td>Property</td>
<td>Property replacement</td>
</tr>
<tr>
<td>=~</td>
<td>String</td>
<td>Regular expression match</td>
</tr>
<tr>
<td>&gt;</td>
<td>Comparison</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;&lt;query-operators-comparison, &gt;⇒&gt;</td>
<td>Comparison</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>AND</td>
<td>Boolean</td>
<td>Conjunction</td>
</tr>
</tbody>
</table>

779
<table>
<thead>
<tr>
<th>Operator</th>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTAINS</td>
<td>String comparison</td>
<td>Case-sensitive inclusion search</td>
</tr>
<tr>
<td>DISTINCT</td>
<td>Aggregation</td>
<td>Duplicate removal</td>
</tr>
<tr>
<td>ENDS WITH</td>
<td>String comparison</td>
<td>Case-sensitive suffix search</td>
</tr>
<tr>
<td>IN</td>
<td>List</td>
<td>List element existence check</td>
</tr>
<tr>
<td>IS NOT NULL</td>
<td>Comparison</td>
<td>Non-null check</td>
</tr>
<tr>
<td>IS NULL</td>
<td>Comparison</td>
<td>null check</td>
</tr>
<tr>
<td>NOT</td>
<td>Boolean</td>
<td>Negation</td>
</tr>
<tr>
<td>OR</td>
<td>Boolean</td>
<td>Disjunction</td>
</tr>
<tr>
<td>STARTS WITH</td>
<td>String comparison</td>
<td>Case-sensitive prefix search</td>
</tr>
<tr>
<td>XOR</td>
<td>Boolean</td>
<td>Exclusive disjunction</td>
</tr>
<tr>
<td>[[]]</td>
<td>Map</td>
<td>Subscript (dynamic value access by key)</td>
</tr>
<tr>
<td>[[]]</td>
<td>Property</td>
<td>Subscript (dynamic property access)</td>
</tr>
<tr>
<td>[[]]</td>
<td>List</td>
<td>Subscript (accessing element(s) in a list)</td>
</tr>
<tr>
<td>^</td>
<td>Mathematical</td>
<td>Exponentiation</td>
</tr>
</tbody>
</table>

### 31.3. Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs()</td>
<td>Numeric</td>
<td>Returns the absolute value of a number.</td>
</tr>
<tr>
<td>acos()</td>
<td>Trigonometric</td>
<td>Returns the arccosine of a number in radians.</td>
</tr>
<tr>
<td>all()</td>
<td>Predicate</td>
<td>Tests whether the predicate holds for all elements in a list.</td>
</tr>
<tr>
<td>any()</td>
<td>Predicate</td>
<td>Tests whether the predicate holds for at least one element in a list.</td>
</tr>
<tr>
<td>asin()</td>
<td>Trigonometric</td>
<td>Returns the arcsine of a number in radians.</td>
</tr>
<tr>
<td>atan()</td>
<td>Trigonometric</td>
<td>Returns the arctangent of a number in radians.</td>
</tr>
<tr>
<td>atan2()</td>
<td>Trigonometric</td>
<td>Returns the arctangent2 of a set of coordinates in radians.</td>
</tr>
<tr>
<td>avg()</td>
<td>Aggregating</td>
<td>Returns the average of a set of values.</td>
</tr>
<tr>
<td>ceil()</td>
<td>Numeric</td>
<td>Returns the smallest floating point number that is greater than or equal to a number and equal to a mathematical integer.</td>
</tr>
<tr>
<td>Function</td>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>coalesce()</td>
<td>Scalar</td>
<td>Returns the first non-null value in a list of expressions.</td>
</tr>
<tr>
<td>collect()</td>
<td>Aggregating</td>
<td>Returns a list containing the values returned by an expression.</td>
</tr>
<tr>
<td>cos()</td>
<td>Trigonometric</td>
<td>Returns the cosine of a number.</td>
</tr>
<tr>
<td>cot()</td>
<td>Trigonometric</td>
<td>Returns the cotangent of a number.</td>
</tr>
<tr>
<td>count()</td>
<td>Aggregating</td>
<td>Returns the number of values or rows.</td>
</tr>
<tr>
<td>date()</td>
<td>Temporal</td>
<td>Returns the current Date.</td>
</tr>
<tr>
<td>date([year [, month, day]])</td>
<td>Temporal</td>
<td>Returns a calendar (Year-Month-Day) Date.</td>
</tr>
<tr>
<td>date([year [, week, dayOfWeek]])</td>
<td>Temporal</td>
<td>Returns a week (Year-Week-Day) Date.</td>
</tr>
<tr>
<td>date([year [, quarter, dayOfQuarter]])</td>
<td>Temporal</td>
<td>Returns a quarter (Year-Quarter-Day) Date.</td>
</tr>
<tr>
<td>date([year [, ordinalDay]])</td>
<td>Temporal</td>
<td>Returns an ordinal (Year-Day) Date.</td>
</tr>
<tr>
<td>date(string)</td>
<td>Temporal</td>
<td>Returns a Date by parsing a string.</td>
</tr>
<tr>
<td>date([map])</td>
<td>Temporal</td>
<td>Returns a Date from a map of another temporal value’s components.</td>
</tr>
<tr>
<td>date.realtime()</td>
<td>Temporal</td>
<td>Returns the current Date using the realtime clock.</td>
</tr>
<tr>
<td>date.statement()</td>
<td>Temporal</td>
<td>Returns the current Date using the statement clock.</td>
</tr>
<tr>
<td>date.transaction()</td>
<td>Temporal</td>
<td>Returns the current Date using the transaction clock.</td>
</tr>
<tr>
<td>date.truncate()</td>
<td>Temporal</td>
<td>Returns a Date obtained by truncating a value at a specific component boundary. Truncation summary.</td>
</tr>
<tr>
<td>datetime()</td>
<td>Temporal</td>
<td>Returns the current DateTime.</td>
</tr>
<tr>
<td>datetime([year [, month, day, ...]])</td>
<td>Temporal</td>
<td>Returns a calendar (Year-Month-Day) DateTime.</td>
</tr>
<tr>
<td>datetime([year [, week, dayOfWeek, ...]])</td>
<td>Temporal</td>
<td>Returns a week (Year-Week-Day) DateTime.</td>
</tr>
<tr>
<td>datetime([year [, quarter, dayOfQuarter, ...]])</td>
<td>Temporal</td>
<td>Returns a quarter (Year-Quarter-Day) DateTime.</td>
</tr>
<tr>
<td>datetime([year [, ordinalDay, ...]])</td>
<td>Temporal</td>
<td>Returns an ordinal (Year-Day) DateTime.</td>
</tr>
<tr>
<td>datetime(string)</td>
<td>Temporal</td>
<td>Returns a DateTime by parsing a string.</td>
</tr>
<tr>
<td>datetime([map])</td>
<td>Temporal</td>
<td>Returns a DateTime from a map of another temporal value’s components.</td>
</tr>
<tr>
<td>Function</td>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>datetime({epochSeconds})</code></td>
<td>Temporal</td>
<td>Returns a DateTime from a timestamp.</td>
</tr>
<tr>
<td><code>datetime.realtime()</code></td>
<td>Temporal</td>
<td>Returns the current DateTime using the <code>realtime</code> clock.</td>
</tr>
<tr>
<td><code>datetime.statement()</code></td>
<td>Temporal</td>
<td>Returns the current DateTime using the <code>statement</code> clock.</td>
</tr>
<tr>
<td><code>datetime.transaction()</code></td>
<td>Temporal</td>
<td>Returns the current DateTime using the <code>transaction</code> clock.</td>
</tr>
<tr>
<td><code>datetime.truncate()</code></td>
<td>Temporal</td>
<td>Returns a DateTime obtained by truncating a value at a specific component boundary. Truncation summary.</td>
</tr>
<tr>
<td><code>degrees()</code></td>
<td>Trigonometric</td>
<td>Converts radians to degrees.</td>
</tr>
<tr>
<td><code>duration({map})</code></td>
<td>Temporal</td>
<td>Returns a Duration from a map of its components.</td>
</tr>
<tr>
<td><code>duration(string)</code></td>
<td>Temporal</td>
<td>Returns a Duration by parsing a string.</td>
</tr>
<tr>
<td><code>duration.between()</code></td>
<td>Temporal</td>
<td>Returns a Duration equal to the difference between two given instants.</td>
</tr>
<tr>
<td><code>duration.inDays()</code></td>
<td>Temporal</td>
<td>Returns a Duration equal to the difference in whole days or weeks between two given instants.</td>
</tr>
<tr>
<td><code>duration.inMonths()</code></td>
<td>Temporal</td>
<td>Returns a Duration equal to the difference in whole months, quarters or years between two given instants.</td>
</tr>
<tr>
<td><code>duration.inSeconds()</code></td>
<td>Temporal</td>
<td>Returns a Duration equal to the difference in seconds and fractions of seconds, or minutes or hours, between two given instants.</td>
</tr>
<tr>
<td>e()</td>
<td>Logarithmic</td>
<td>Returns the base of the natural logarithm, e.</td>
</tr>
<tr>
<td><code>endNode()</code></td>
<td>Scalar</td>
<td>Returns the end node of a relationship.</td>
</tr>
<tr>
<td><code>exists()</code></td>
<td>Predicate</td>
<td>Returns true if a match for the pattern exists in the graph, or if the specified property exists in the node, relationship or map.</td>
</tr>
<tr>
<td>exp()</td>
<td>Logarithmic</td>
<td>Returns e^n, where e is the base of the natural logarithm, and n is the value of the argument expression.</td>
</tr>
<tr>
<td>floor()</td>
<td>Numeric</td>
<td>Returns the largest floating point number that is less than or equal to a number and equal to a mathematical integer.</td>
</tr>
<tr>
<td>haversin()</td>
<td>Trigonometric</td>
<td>Returns half the versine of a number.</td>
</tr>
<tr>
<td>Function</td>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>head()</code></td>
<td>Scalar</td>
<td>Returns the first element in a list.</td>
</tr>
<tr>
<td><code>id()</code></td>
<td>Scalar</td>
<td>Returns the id of a relationship or node.</td>
</tr>
<tr>
<td><code>isEmpty()</code></td>
<td>Predicate</td>
<td>Returns true if the given list or map contains no elements or if the given string contains no characters.</td>
</tr>
<tr>
<td><code>keys()</code></td>
<td>List</td>
<td>Returns a list containing the string representations for all the property names of a node, relationship, or map.</td>
</tr>
<tr>
<td><code>labels()</code></td>
<td>List</td>
<td>Returns a list containing the string representations for all the labels of a node.</td>
</tr>
<tr>
<td><code>last()</code></td>
<td>Scalar</td>
<td>Returns the last element in a list.</td>
</tr>
<tr>
<td><code>left()</code></td>
<td>String</td>
<td>Returns a string containing the specified number of leftmost characters of the original string.</td>
</tr>
<tr>
<td><code>length()</code></td>
<td>Scalar</td>
<td>Returns the length of a path.</td>
</tr>
<tr>
<td><code>localdatetime()</code></td>
<td>Temporal</td>
<td>Returns the current LocalDateTime.</td>
</tr>
<tr>
<td><code>localdatetime(year [, month, day, ...])</code></td>
<td>Temporal</td>
<td>Returns a calendar (Year-Month-Day) LocalDateTime.</td>
</tr>
<tr>
<td><code>localdatetime(year [, week, dayOfWeek, ...])</code></td>
<td>Temporal</td>
<td>Returns a week (Year-Week-Day) LocalDateTime.</td>
</tr>
<tr>
<td><code>localdatetime(year [, quarter, dayOfQuarter, ...])</code></td>
<td>Temporal</td>
<td>Returns a quarter (Year-Quarter-Day) DateTime.</td>
</tr>
<tr>
<td><code>localdatetime(year [, ordinalDay, ...])</code></td>
<td>Temporal</td>
<td>Returns an ordinal (Year-Day) LocalDateTime.</td>
</tr>
<tr>
<td><code>localdatetime(string)</code></td>
<td>Temporal</td>
<td>Returns a LocalDateTime by parsing a string.</td>
</tr>
<tr>
<td><code>localdatetime(map)</code></td>
<td>Temporal</td>
<td>Returns a LocalDateTime from a map of another temporal value’s components.</td>
</tr>
<tr>
<td><code>localdatetime.realtime()</code></td>
<td>Temporal</td>
<td>Returns the current LocalDateTime using the <code>realtime</code> clock.</td>
</tr>
<tr>
<td><code>localdatetime.statement()</code></td>
<td>Temporal</td>
<td>Returns the current LocalDateTime using the <code>statement</code> clock.</td>
</tr>
<tr>
<td><code>localdatetime.transaction()</code></td>
<td>Temporal</td>
<td>Returns the current LocalDateTime using the <code>transaction</code> clock.</td>
</tr>
<tr>
<td><code>localdatetime.truncate()</code></td>
<td>Temporal</td>
<td>Returns a LocalDateTime obtained by truncating a value at a specific component boundary. Truncation summary.</td>
</tr>
<tr>
<td><code>localtime()</code></td>
<td>Temporal</td>
<td>Returns the current LocalTime.</td>
</tr>
<tr>
<td>Function</td>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>localtime(hour [, minute, second, ...])</code></td>
<td>Temporal</td>
<td>Returns a LocalTime with the specified component values.</td>
</tr>
<tr>
<td><code>localtime(string)</code></td>
<td>Temporal</td>
<td>Returns a LocalTime by parsing a string.</td>
</tr>
<tr>
<td><code>localtime(time [, hour, ...])</code></td>
<td>Temporal</td>
<td>Returns a LocalTime from a map of another temporal value’s components.</td>
</tr>
<tr>
<td><code>localtime.realtime()</code></td>
<td>Temporal</td>
<td>Returns the current LocalTime using the <code>realtime</code> clock.</td>
</tr>
<tr>
<td><code>localtime.statement()</code></td>
<td>Temporal</td>
<td>Returns the current LocalTime using the <code>statement</code> clock.</td>
</tr>
<tr>
<td><code>localtime.transaction()</code></td>
<td>Temporal</td>
<td>Returns the current LocalTime using the <code>transaction</code> clock.</td>
</tr>
<tr>
<td><code>localtime.truncate()</code></td>
<td>Temporal</td>
<td>Returns a LocalTime obtained by truncating a value at a specific component boundary. Truncation summary.</td>
</tr>
<tr>
<td><code>log()</code></td>
<td>Logarithmic</td>
<td>Returns the natural logarithm of a number.</td>
</tr>
<tr>
<td><code>log10()</code></td>
<td>Logarithmic</td>
<td>Returns the common logarithm (base 10) of a number.</td>
</tr>
<tr>
<td><code>lTrim()</code></td>
<td>String</td>
<td>Returns the original string with leading whitespace removed.</td>
</tr>
<tr>
<td><code>max()</code></td>
<td>Aggregating</td>
<td>Returns the maximum value in a set of values.</td>
</tr>
<tr>
<td><code>min()</code></td>
<td>Aggregating</td>
<td>Returns the minimum value in a set of values.</td>
</tr>
<tr>
<td><code>nodes()</code></td>
<td>List</td>
<td>Returns a list containing all the nodes in a path.</td>
</tr>
<tr>
<td><code>none()</code></td>
<td>Predicate</td>
<td>Returns true if the predicate holds for no element in a list.</td>
</tr>
<tr>
<td><code>percentileCont()</code></td>
<td>Aggregating</td>
<td>Returns the percentile of the given value over a group using linear interpolation.</td>
</tr>
<tr>
<td><code>percentileDisc()</code></td>
<td>Aggregating</td>
<td>Returns the nearest value to the given percentile over a group using a rounding method.</td>
</tr>
<tr>
<td><code>pi()</code></td>
<td>Trigonometric</td>
<td>Returns the mathematical constant pi.</td>
</tr>
<tr>
<td><code>point() - Cartesian 2D</code></td>
<td>Spatial</td>
<td>Returns a 2D point object, given two coordinate values in the Cartesian coordinate system.</td>
</tr>
<tr>
<td><code>point() - Cartesian 3D</code></td>
<td>Spatial</td>
<td>Returns a 3D point object, given three coordinate values in the Cartesian coordinate system.</td>
</tr>
<tr>
<td>Function</td>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>point() - WGS 84 2D</td>
<td>Spatial</td>
<td>Returns a 2D point object, given two coordinate values in the WGS 84 coordinate system.</td>
</tr>
<tr>
<td>point() - WGS 84 3D</td>
<td>Spatial</td>
<td>Returns a 3D point object, given three coordinate values in the WGS 84 coordinate system.</td>
</tr>
<tr>
<td>point.distance()</td>
<td>Spatial</td>
<td>Returns true if the provided point is within the bounding box defined by the two provided points.</td>
</tr>
<tr>
<td>point.withinBBox()</td>
<td>Spatial</td>
<td>Returns a floating point number representing the geodesic distance between any two points in the same CRS.</td>
</tr>
<tr>
<td>properties()</td>
<td>Scalar</td>
<td>Returns a map containing all the properties of a node or relationship.</td>
</tr>
<tr>
<td>radians()</td>
<td>Trigonometric</td>
<td>Converts degrees to radians.</td>
</tr>
<tr>
<td>rand()</td>
<td>Numeric</td>
<td>Returns a random floating point number in the range from 0 (inclusive) to 1 (exclusive); i.e. [0, 1).</td>
</tr>
<tr>
<td>randomUUID()</td>
<td>Scalar</td>
<td>Returns a string value corresponding to a randomly-generated UUID.</td>
</tr>
<tr>
<td>range()</td>
<td>List</td>
<td>Returns a list comprising all integer values within a specified range.</td>
</tr>
<tr>
<td>reduce()</td>
<td>List</td>
<td>Runs an expression against individual elements of a list, storing the result of the expression in an accumulator.</td>
</tr>
<tr>
<td>relationships()</td>
<td>List</td>
<td>Returns a list containing all the relationships in a path.</td>
</tr>
<tr>
<td>replace()</td>
<td>String</td>
<td>Returns a string in which all occurrences of a specified string in the original string have been replaced by another (specified) string.</td>
</tr>
<tr>
<td>reverse()</td>
<td>List</td>
<td>Returns a list in which the order of all elements in the original list have been reversed.</td>
</tr>
<tr>
<td>reverse()</td>
<td>String</td>
<td>Returns a string in which the order of all characters in the original string have been reversed.</td>
</tr>
<tr>
<td>right()</td>
<td>String</td>
<td>Returns a string containing the specified number of rightmost characters of the original string.</td>
</tr>
<tr>
<td>Function</td>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>--------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>round()</td>
<td>Numeric</td>
<td>Returns the value of the given number rounded to the nearest integer, with half-way values always rounded up.</td>
</tr>
<tr>
<td>round(), with precision</td>
<td>Numeric</td>
<td>Returns the value of the given number rounded with the specified precision, with half-values always being rounded up.</td>
</tr>
<tr>
<td>round(), with precision and rounding mode</td>
<td>Numeric</td>
<td>Returns the value of the given number rounded with the specified precision and the specified rounding mode.</td>
</tr>
<tr>
<td>rTrim()</td>
<td>String</td>
<td>Returns the original string with trailing whitespace removed.</td>
</tr>
<tr>
<td>sign()</td>
<td>Numeric</td>
<td>Returns the signum of a number: 0 if the number is 0, -1 for any negative number, and 1 for any positive number.</td>
</tr>
<tr>
<td>sin()</td>
<td>Trigonometric</td>
<td>Returns the sine of a number.</td>
</tr>
<tr>
<td>single()</td>
<td>Predicate</td>
<td>Returns true if the predicate holds for exactly one of the elements in a list.</td>
</tr>
<tr>
<td>size()</td>
<td>Scalar</td>
<td>Returns the number of items in a list.</td>
</tr>
<tr>
<td>size() applied to pattern comprehension</td>
<td>Scalar</td>
<td>Returns the number of paths matching the pattern comprehension.</td>
</tr>
<tr>
<td>size() applied to string</td>
<td>Scalar</td>
<td>Returns the number of Unicode characters in a string.</td>
</tr>
<tr>
<td>split()</td>
<td>String</td>
<td>Returns a list of strings resulting from the splitting of the original string around matches of the given delimiter.</td>
</tr>
<tr>
<td>sqrt()</td>
<td>Logarithmic</td>
<td>Returns the square root of a number.</td>
</tr>
<tr>
<td>startNode()</td>
<td>Scalar</td>
<td>Returns the start node of a relationship.</td>
</tr>
<tr>
<td>stDev()</td>
<td>Aggregating</td>
<td>Returns the standard deviation for the given value over a group for a sample of a population.</td>
</tr>
<tr>
<td>stDevP()</td>
<td>Aggregating</td>
<td>Returns the standard deviation for the given value over a group for an entire population.</td>
</tr>
<tr>
<td>substring()</td>
<td>String</td>
<td>Returns a substring of the original string, beginning with a 0-based index start and length.</td>
</tr>
<tr>
<td>sum()</td>
<td>Aggregating</td>
<td>Returns the sum of a set of numeric values.</td>
</tr>
<tr>
<td>tail()</td>
<td>List</td>
<td>Returns all but the first element in a list.</td>
</tr>
<tr>
<td>tan()</td>
<td>Trigonometric</td>
<td>Returns the tangent of a number.</td>
</tr>
<tr>
<td>Function</td>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>time()</td>
<td>Temporal</td>
<td>Returns the current Time.</td>
</tr>
<tr>
<td>time([hour [, minute, ...]])</td>
<td>Temporal</td>
<td>Returns a Time with the specified component values.</td>
</tr>
<tr>
<td>time(string)</td>
<td>Temporal</td>
<td>Returns a Time by parsing a string.</td>
</tr>
<tr>
<td>time({time [, hour, ..., timezone]})</td>
<td>Temporal</td>
<td>Returns a Time from a map of another temporal value’s components.</td>
</tr>
<tr>
<td>time.realtime()</td>
<td>Temporal</td>
<td>Returns the current Time using the realtime clock.</td>
</tr>
<tr>
<td>time.statement()</td>
<td>Temporal</td>
<td>Returns the current Time using the statement clock.</td>
</tr>
<tr>
<td>time.transaction()</td>
<td>Temporal</td>
<td>Returns the current Time using the transaction clock.</td>
</tr>
<tr>
<td>time.truncate()</td>
<td>Temporal</td>
<td>Returns a Time obtained by truncating a value at a specific component boundary. Truncation summary.</td>
</tr>
<tr>
<td>timestamp()</td>
<td>Scalar</td>
<td>Returns the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC.</td>
</tr>
<tr>
<td>toBoolean()</td>
<td>Scalar</td>
<td>Converts a string value to a boolean value.</td>
</tr>
<tr>
<td>toFloat()</td>
<td>Scalar</td>
<td>Converts an integer or string value to a floating point number.</td>
</tr>
<tr>
<td>toInteger()</td>
<td>Scalar</td>
<td>Converts a floating point or string value to an integer value.</td>
</tr>
<tr>
<td>toLower()</td>
<td>String</td>
<td>Returns the original string in lowercase.</td>
</tr>
<tr>
<td>toString()</td>
<td>String</td>
<td>Converts an integer, float, boolean or temporal (i.e. Date, Time, LocalTime, DateTime, LocalDateTime or Duration) value to a string.</td>
</tr>
<tr>
<td>toUpper()</td>
<td>String</td>
<td>Returns the original string in uppercase.</td>
</tr>
<tr>
<td>trim()</td>
<td>String</td>
<td>Returns the original string with leading and trailing whitespace removed.</td>
</tr>
<tr>
<td>type()</td>
<td>Scalar</td>
<td>Returns the string representation of the relationship type.</td>
</tr>
</tbody>
</table>

31.4. Expressions

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASE Expression</td>
<td>A generic conditional expression, similar to if/else statements available in other languages.</td>
</tr>
</tbody>
</table>
31.5. Cypher query options

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CYPHER $version query</td>
<td>Version</td>
<td>This will force <code>query</code> to use Neo4j Cypher $version. The default is 4.0.</td>
</tr>
<tr>
<td>CYPHER runtime=interpreted query</td>
<td>Runtime</td>
<td>This will force the query planner to use the interpreted runtime. This is the only option in Neo4j Community Edition.</td>
</tr>
<tr>
<td>CYPHER runtime=slotted query</td>
<td>Runtime</td>
<td>This will cause the query planner to use the slotted runtime. This is only available in Neo4j Enterprise Edition.</td>
</tr>
<tr>
<td>CYPHER runtime=pipelined query</td>
<td>Runtime</td>
<td>This will cause the query planner to use the pipelined runtime if it supports <code>query</code>. This is only available in Neo4j Enterprise Edition.</td>
</tr>
</tbody>
</table>

31.6. Administrative commands

The following commands are only executable against the system database:

<table>
<thead>
<tr>
<th>Command</th>
<th>Admin category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTER ALIAS ... [IF EXISTS] SET DATABASE ...</td>
<td>Database alias</td>
<td>Modifies a database alias.</td>
</tr>
<tr>
<td>ALTER CURRENT USER SET PASSWORD FROM ... TO</td>
<td>User and role</td>
<td>Change the password of the user that is currently logged in.</td>
</tr>
<tr>
<td>ALTER DATABASE ... [IF EXISTS] SET ACCESS {READ ONLY</td>
<td>READ WRITE}</td>
<td>Database</td>
</tr>
<tr>
<td>ALTER USER ... [IF EXISTS] [SET [PLAINTEXT</td>
<td>ENCRYPTED] PASSWORD [password [CHANGE [NOT] REQUIRED]</td>
<td>CHANGE [NOT] REQUIRED] [SET STATUS {ACTIVE</td>
</tr>
<tr>
<td>CREATE [OR REPLACE] ALIAS ... [IF NOT EXISTS] FOR DATABASE ...</td>
<td>Database alias</td>
<td>Creates a new database alias.</td>
</tr>
<tr>
<td>CREATE [OR REPLACE] DATABASE ... [IF NOT EXISTS] [OPTIONS {optionKey : optionValue[, ...]}] [WAIT [n [SEC[OND[S]]]]] NOWAIT]</td>
<td>Database</td>
<td>Creates a new database.</td>
</tr>
<tr>
<td>CREATE [OR REPLACE] ROLE ... [IF NOT EXISTS] [AS COPY OF]</td>
<td>User and role</td>
<td>Creates new roles.</td>
</tr>
<tr>
<td>Command</td>
<td>Admin category</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>**CREATE [OR REPLACE] USER ... [IF NOT EXISTS] SET [PLAINTEXT</td>
<td>ENCRYPTED] PASSWORD ... [SET PASSWORD] CHANGE [NOT] REQUIRED [SET STATUS {ACTIVE</td>
<td>SUSPENDED}] [SET HOME DATABASE name]]**</td>
</tr>
<tr>
<td><strong>DENY ... ON DATABASE ... TO</strong></td>
<td>Privilege</td>
<td>Denies a database or schema privilege to one or multiple roles.</td>
</tr>
<tr>
<td><strong>DENY ... ON DBMS TO</strong></td>
<td>Privilege</td>
<td>Denies a DBMS privilege to one or multiple roles.</td>
</tr>
<tr>
<td>**DENY ... ON GRAPH ... [NODES</td>
<td>RELATIONSHIPS</td>
<td>ELEMENTS] ... TO**</td>
</tr>
<tr>
<td><strong>DROP ALIAS ... [IF EXISTS] FOR DATABASE</strong></td>
<td>Database alias</td>
<td>Deletes a specified database alias.</td>
</tr>
<tr>
<td>**DROP DATABASE ... [IF EXISTS] [DUMP DATA</td>
<td>DESTROY DATA]**</td>
<td>Database</td>
</tr>
<tr>
<td><strong>DROP ROLE ... [IF EXISTS]</strong></td>
<td>User and role</td>
<td>Deletes a specified role.</td>
</tr>
<tr>
<td><strong>DROP USER ... [IF EXISTS]</strong></td>
<td>User and role</td>
<td>Deletes a specified user.</td>
</tr>
<tr>
<td><strong>GRANT ... ON DATABASE ... TO</strong></td>
<td>Privilege</td>
<td>Assigns a database or schema privilege to one or multiple roles.</td>
</tr>
<tr>
<td><strong>GRANT ... ON DBMS TO</strong></td>
<td>Privilege</td>
<td>Assigns a DBMS privilege to one or multiple roles.</td>
</tr>
<tr>
<td>**GRANT ... ON GRAPH ... [NODES</td>
<td>RELATIONSHIPS</td>
<td>ELEMENTS] ... TO**</td>
</tr>
<tr>
<td><strong>GRANT ROLE[S] ... TO</strong></td>
<td>User and role</td>
<td>Assigns one or multiple roles to one or multiple users.</td>
</tr>
<tr>
<td><strong>RENAME ROLE ... [IF EXISTS] TO ...</strong></td>
<td>User and role</td>
<td>Changes the name of a role.</td>
</tr>
<tr>
<td><strong>RENAME USER ... [IF EXISTS] TO ...</strong></td>
<td>User and role</td>
<td>Changes the name of a user.</td>
</tr>
<tr>
<td>**REVOKE [GRANT</td>
<td>DENY] ... ON DATABASE ... FROM**</td>
<td>Privilege</td>
</tr>
<tr>
<td>**REVOKE [GRANT</td>
<td>DENY] ... ON DBMS FROM**</td>
<td>Privilege</td>
</tr>
<tr>
<td>**REVOKE [GRANT</td>
<td>DENY] ... ON GRAPH ... [NODES</td>
<td>RELATIONSHIPS</td>
</tr>
<tr>
<td><strong>REVOKE ROLE[S] ... FROM</strong></td>
<td>User and role</td>
<td>Removes one or multiple roles from one or multiple users.</td>
</tr>
<tr>
<td>Command</td>
<td>Admin category</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>SHOW ALIASES FOR DATABASE</td>
<td>Database alias</td>
<td>Returns information about all aliases, optionally including driver settings.</td>
</tr>
<tr>
<td>SHOW [ALL</td>
<td>POPULATED] ROLES [WITH USERS]]</td>
<td>User and role</td>
</tr>
<tr>
<td>SHOW DATABASE</td>
<td>Database</td>
<td>Returns information about a specified database.</td>
</tr>
<tr>
<td>SHOW DATABASES</td>
<td>Database</td>
<td>Returns information about all databases.</td>
</tr>
<tr>
<td>SHOW DEFAULT DATABASE</td>
<td>Database</td>
<td>Returns information about the default database.</td>
</tr>
<tr>
<td>SHOW HOME DATABASE</td>
<td>Database</td>
<td>Returns information about the current users home database.</td>
</tr>
<tr>
<td>SHOW [ROLE ...</td>
<td>USER ...</td>
<td>ALL \ PRIVILEGES [AS [REVOKE] COMMAND[S]\]]</td>
</tr>
<tr>
<td>SHOW USERS</td>
<td>User and role</td>
<td>Returns information about all users.</td>
</tr>
<tr>
<td>START DATABASE</td>
<td>Database</td>
<td>Starts up a specified database.</td>
</tr>
<tr>
<td>STOP DATABASE</td>
<td>Database</td>
<td>Stops a specified database.</td>
</tr>
</tbody>
</table>

### 31.7. Privilege Actions

<table>
<thead>
<tr>
<th>Name</th>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS</td>
<td>Database</td>
<td>Determines whether a user can access a specific database.</td>
</tr>
<tr>
<td>ALL DATABASE PRIVILEGES</td>
<td>Database and schema</td>
<td>Determines whether a user is allowed to access, create, drop, and list indexes and constraints, create new labels, types and property names on a specific database.</td>
</tr>
<tr>
<td>ALL DBMS PRIVILEGES</td>
<td>DBMS</td>
<td>Determines whether a user is allowed to perform role, user, database and privilege management.</td>
</tr>
<tr>
<td>ALL GRAPH PRIVILEGES</td>
<td>GRAPH</td>
<td>Determines whether a user is allowed to perform reads and writes.</td>
</tr>
<tr>
<td>ALTER ALIAS</td>
<td>DBMS</td>
<td>Determines whether the user can modify aliases.</td>
</tr>
<tr>
<td>ALTER DATABASE</td>
<td>DBMS</td>
<td>Determines whether the user can modify databases and aliases.</td>
</tr>
<tr>
<td>ALTER USER</td>
<td>DBMS</td>
<td>Determines whether the user can modify users.</td>
</tr>
<tr>
<td>Name</td>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ASSIGN PRIVILEGE</td>
<td>DBMS</td>
<td>Determines whether the user can assign privileges using the GRANT and DENY commands.</td>
</tr>
<tr>
<td>ASSIGN ROLE</td>
<td>DBMS</td>
<td>Determines whether the user can grant roles.</td>
</tr>
<tr>
<td>CONSTRAINT MANAGEMENT</td>
<td>Schema</td>
<td>Determines whether a user is allowed to create, drop, and list constraints on a specific database.</td>
</tr>
<tr>
<td>CREATE</td>
<td>GRAPH</td>
<td>Determines whether the user can create a new element (node, relationship or both).</td>
</tr>
<tr>
<td>CREATE ALIAS</td>
<td>DBMS</td>
<td>Determines whether the user can create new aliases.</td>
</tr>
<tr>
<td>CREATE CONSTRAINT</td>
<td>Schema</td>
<td>Determines whether a user is allowed to create constraints on a specific database.</td>
</tr>
<tr>
<td>CREATE DATABASE</td>
<td>DBMS</td>
<td>Determines whether the user can create new databases and aliases.</td>
</tr>
<tr>
<td>CREATE INDEX</td>
<td>Schema</td>
<td>Determines whether a user is allowed to create indexes on a specific database.</td>
</tr>
<tr>
<td>CREATE NEW NODE LABEL</td>
<td>Schema</td>
<td>Determines whether a user is allowed to create new node labels on a specific database.</td>
</tr>
<tr>
<td>CREATE NEW PROPERTY NAME</td>
<td>Schema</td>
<td>Determines whether a user is allowed to create new property names on a specific database.</td>
</tr>
<tr>
<td>CREATE NEW RELATIONSHIP TYPE</td>
<td>Schema</td>
<td>Determines whether a user is allowed to create new relationship types on a specific database.</td>
</tr>
<tr>
<td>CREATE ROLE</td>
<td>DBMS</td>
<td>Determines whether the user can create new roles.</td>
</tr>
<tr>
<td>CREATE USER</td>
<td>DBMS</td>
<td>Determines whether the user can create new users.</td>
</tr>
<tr>
<td>ALIAS MANAGEMENT</td>
<td>DBMS</td>
<td>Determines whether the user can create, delete, modify and list aliases.</td>
</tr>
<tr>
<td>DATABASE MANAGEMENT</td>
<td>DBMS</td>
<td>Determines whether the user can create, delete, and modify databases and aliases.</td>
</tr>
<tr>
<td>DELETE</td>
<td>GRAPH</td>
<td>Determines whether the user can delete an element (node, relationship or both).</td>
</tr>
<tr>
<td>DROP ALIAS</td>
<td>DBMS</td>
<td>Determines whether the user can delete aliases.</td>
</tr>
<tr>
<td>Name</td>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DROP CONSTRAINT</td>
<td>Schema</td>
<td>Determines whether a user is allowed to drop constraints on a specific database.</td>
</tr>
<tr>
<td>DROP DATABASE</td>
<td>DBMS</td>
<td>Determines whether the user can delete databases and aliases.</td>
</tr>
<tr>
<td>DROP INDEX</td>
<td>Schema</td>
<td>Determines whether a user is allowed to drop indexes on a specific database.</td>
</tr>
<tr>
<td>DROP ROLE</td>
<td>DBMS</td>
<td>Determines whether the user can delete roles.</td>
</tr>
<tr>
<td>DROP USER</td>
<td>DBMS</td>
<td>Determines whether the user can delete users.</td>
</tr>
<tr>
<td>EXECUTE ADMIN PROCEDURE</td>
<td>DBMS</td>
<td>Determines whether the user can execute admin procedures.</td>
</tr>
<tr>
<td>EXECUTE BOOSTED FUNCTION</td>
<td>DBMS</td>
<td>Determines whether the user can execute functions with elevated privileges.</td>
</tr>
<tr>
<td>EXECUTE BOOSTED PROCEDURE</td>
<td>DBMS</td>
<td>Determines whether the user can execute procedures with elevated privileges.</td>
</tr>
<tr>
<td>EXECUTE FUNCTION</td>
<td>DBMS</td>
<td>Determines whether the user can execute functions.</td>
</tr>
<tr>
<td>EXECUTE PROCEDURE</td>
<td>DBMS</td>
<td>Determines whether the user can execute procedures.</td>
</tr>
<tr>
<td>IMPERSONATE</td>
<td>DBMS</td>
<td>Determines whether a user can impersonate another one and assume their privileges.</td>
</tr>
<tr>
<td>INDEX MANAGEMENT</td>
<td>Schema</td>
<td>Determines whether a user is allowed to create, drop, and list indexes on a specific database.</td>
</tr>
<tr>
<td>MATCH</td>
<td>GRAPH</td>
<td>Determines whether the properties of an element (node, relationship or both) can be read and the element can be found and traversed while executing queries on the specified graph.</td>
</tr>
<tr>
<td>MERGE</td>
<td>GRAPH</td>
<td>Determines whether the user can find, read, create and set properties on an element (node, relationship or both).</td>
</tr>
<tr>
<td>NAME MANAGEMENT</td>
<td>Schema</td>
<td>Determines whether a user is allowed to create new labels, types and property names on a specific database.</td>
</tr>
<tr>
<td>PRIVILEGE MANAGEMENT</td>
<td>DBMS</td>
<td>Determines whether the user can show, assign and remove privileges.</td>
</tr>
<tr>
<td>Name</td>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>READ</td>
<td>GRAPH</td>
<td>Determines whether the properties of an element (node, relationship or both) can be read while executing queries on the specified graph.</td>
</tr>
<tr>
<td>REMOVE LABEL</td>
<td>GRAPH</td>
<td>Determines whether the user can remove a label from a node using the REMOVE clause.</td>
</tr>
<tr>
<td>REMOVE PRIVILEGE</td>
<td>DBMS</td>
<td>Determines whether the user can remove privileges using the REVOKE command.</td>
</tr>
<tr>
<td>REMOVE ROLE</td>
<td>DBMS</td>
<td>Determines whether the user can revoke roles.</td>
</tr>
<tr>
<td>RENAME ROLE</td>
<td>DBMS</td>
<td>Determines whether the user can rename roles.</td>
</tr>
<tr>
<td>RENAME USER</td>
<td>DBMS</td>
<td>Determines whether the user can rename users.</td>
</tr>
<tr>
<td>ROLE MANAGEMENT</td>
<td>DBMS</td>
<td>Determines whether the user can create, drop, grant, revoke and show roles.</td>
</tr>
<tr>
<td>SET DATABASE ACCESS</td>
<td>DBMS</td>
<td>Determines whether the user can modify the database access mode.</td>
</tr>
<tr>
<td>SET LABEL</td>
<td>GRAPH</td>
<td>Determines whether the user can set a label to a node using the SET clause.</td>
</tr>
<tr>
<td>SET PASSWORDS</td>
<td>DBMS</td>
<td>Determines whether the user can modify users' passwords and whether those passwords must be changed upon first login.</td>
</tr>
<tr>
<td>SET PROPERTY</td>
<td>GRAPH</td>
<td>Determines whether the user can set a property to an element (node, relationship or both) using the SET clause.</td>
</tr>
<tr>
<td>SET USER HOME DATABASE</td>
<td>DBMS</td>
<td>Determines whether the user can modify the home database of users.</td>
</tr>
<tr>
<td>SET USER STATUS</td>
<td>DBMS</td>
<td>Determines whether the user can modify the account status of users. Adamitersity.</td>
</tr>
<tr>
<td>SHOW ALIAS</td>
<td>DBMS</td>
<td>Determines whether the user is allowed to list aliases.</td>
</tr>
<tr>
<td>SHOW CONSTRAINT</td>
<td>Schema</td>
<td>Determines whether the user is allowed to list constraints.</td>
</tr>
<tr>
<td>SHOW INDEX</td>
<td>Schema</td>
<td>Determines whether the user is allowed to list indexes.</td>
</tr>
<tr>
<td>Name</td>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SHOW PRIVILEGE</td>
<td>DBMS</td>
<td>Determines whether the user can get information about privileges assigned to users and roles.</td>
</tr>
<tr>
<td>SHOW ROLE</td>
<td>DBMS</td>
<td>Determines whether the user can get information about existing and assigned roles.</td>
</tr>
<tr>
<td>SHOW TRANSACTION</td>
<td>Database</td>
<td>Determines whether a user is allowed to list transactions and queries.</td>
</tr>
<tr>
<td>SHOW USER</td>
<td>DBMS</td>
<td>Determines whether the user can get information about existing users.</td>
</tr>
<tr>
<td>START</td>
<td>Database</td>
<td>Determines whether a user can start up a specific database.</td>
</tr>
<tr>
<td>STOP</td>
<td>Database</td>
<td>Determines whether a user can stop a specific running database.</td>
</tr>
<tr>
<td>TERMINATE TRANSACTION</td>
<td>Database</td>
<td>Determines whether a user is allowed to end running transactions and queries.</td>
</tr>
<tr>
<td>TRANSACTION MANAGEMENT</td>
<td>Database</td>
<td>Determines whether a user is allowed to list and end running transactions and queries.</td>
</tr>
<tr>
<td>TRAVERSE</td>
<td>GRAPH</td>
<td>Determines whether an element (node, relationship or both) can be found and traversed while executing queries on the specified graph.</td>
</tr>
<tr>
<td>USER MANAGEMENT</td>
<td>DBMS</td>
<td>Determines whether the user can create, drop, modify and show users.</td>
</tr>
<tr>
<td>WRITE</td>
<td>GRAPH</td>
<td>Determines whether the user can execute write operations on the specified graph.</td>
</tr>
</tbody>
</table>
Appendix A: Cypher styleguide

This appendix contains the recommended style when writing Cypher queries.

This appendix contains the following:

- General recommendations
- Indentations and line breaks
- Casing
- Spacing
- Patterns
- Meta characters

The purpose of the styleguide is to make the code as easy to read as possible, and thereby contributing to lower cost of maintenance.

For rules and recommendations for naming of labels, relationship types and properties, please see the Naming rules and recommendations.

A.1. General recommendations

- When using Cypher language constructs in prose, use a monospaced font and follow the styling rules.
- When referring to labels and relationship types, the colon should be included as follows: :Label, :REL_TYPE.
- When referring to functions, use lower camel case and parentheses should be used as follows: shortestPath(). Arguments should normally not be included.
- If you are storing Cypher statements in a separate file, use the file extension .cypher.

A.2. Indentation and line breaks

- Start a new clause on a new line.

  Bad

  ```cypher
  MATCH (n) WHERE n.name CONTAINS 's' RETURN n.name
  ```

  Good

  ```cypher
  MATCH (n)
  WHERE n.name CONTAINS 's'
  RETURN n.name
  ```

- Indent ON CREATE and ON MATCH with two spaces. Put ON CREATE before ON MATCH if both are present.
Bad

```
MERGE (n) ON CREATE SET n.prop = 0
MERGE (a:A)-[:T]->(b:B)
ON MATCH SET b.name = 'you'
ON CREATE SET a.name = 'me'
RETURN a.prop
```

Good

```
MERGE (n)
  ON CREATE SET n.prop = 0
MERGE (a:A)-[:T]->(b:B)
  ON CREATE SET a.name = 'me'
  ON MATCH SET b.name = 'you'
RETURN a.prop
```

- Start a subquery on a new line after the opening brace, indented with two (additional) spaces. Leave the closing brace on its own line.

Bad

```
MATCH (a:A)
WHERE
  EXISTS { MATCH (a)-->(b:B) WHERE b.prop = $param }
RETURN a.foo
```

Also bad

```
MATCH (a:A)
WHERE EXISTS
  {MATCH (a)-->(b:B)
  WHERE b.prop = $param}
RETURN a.foo
```

Good

```
MATCH (a:A)
WHERE EXISTS {
  MATCH (a)-->(b:B)
  WHERE b.prop = $param
}
RETURN a.foo
```

- Do not break the line if the simplified subquery form is used.

Bad

```
MATCH (a:A)
WHERE EXISTS {
  (a)-->(b:B)
}
RETURN a.prop
```

Good

```
MATCH (a:A)
WHERE EXISTS { (a)-->(b:B) }
RETURN a.prop
```
A.3. Casing

- Write keywords in upper case.

**Bad**

```cypher
match (p:Person)
where p.name starts with 'Ma'
return p.name
```

**Good**

```cypher
MATCH (p:Person)
WHERE p.name STARTS WITH 'Ma'
RETURN p.name
```

- Write the value `null` in lower case.

**Bad**

```cypher
WITH NULL AS n1, Null AS n2
RETURN n1 IS NULL AND n2 IS NOT NULL
```

**Good**

```cypher
WITH null AS n1, null AS n2
RETURN n1 IS NULL AND n2 IS NOT NULL
```

- Write boolean literals (`true` and `false`) in lower case.

**Bad**

```cypher
WITH TRUE AS b1, False AS b2
RETURN b1 AND b2
```

**Good**

```cypher
WITH true AS b1, false AS b2
RETURN b1 AND b2
```

- Use camel case, starting with a lower-case character, for:
  - functions
  - properties
  - variables
  - parameters

**Bad**

```cypher
CREATE (N {Prop: 0})
WITH RAND() AS Rand, $pArAm AS MAP
RETURN Rand, MAP.property_key, Count(N)
```
A.4. Spacing

- For literal maps:
  - No space between the opening brace and the first key
  - No space between key and colon
  - One space between colon and value
  - No space between value and comma
  - One space between comma and next key
  - No space between the last value and the closing brace

**Bad**

```plaintext
WITH { key1 : 'value' , key2 : 42 } AS map
RETURN map
```

**Good**

```plaintext
WITH { key1 : 'value' , key2 : 42 } AS map
RETURN map
```

- One space between label/type predicates and property predicates in patterns.

**Bad**

```plaintext
MATCH (p:Person{property: -1})-[[:KNOWS {since: 2016}]->()<)
RETURN p.name
```

**Good**

```plaintext
MATCH (p:Person {property: -1})-[[:KNOWS {since: 2016}]->()<)
RETURN p.name
```

- No space in patterns.

**Bad**

```plaintext
MATCH (:Person) --> (:Vehicle)
RETURN count(*)
```

**Good**

```plaintext
MATCH (:Person)-->(:Vehicle)
RETURN count(*)
```
• Use a wrapping space around operators.

**Bad**

```sql
MATCH p=(s)--(e)
WHERE s.name<>e.name
RETURN length(p)
```

**Good**

```sql
MATCH p = (s)--(e)
WHERE s.name <> e.name
RETURN length(p)
```

• No space in label predicates.

**Bad**

```sql
MATCH (person:Person:Owner)
RETURN person.name
```

**Good**

```sql
MATCH (person:Person:Owner)
RETURN person.name
```

• Use a space after each comma in lists and enumerations.

**Bad**

```sql
MATCH (),()
WITH ['a','b',3.14] AS list
RETURN list,2,3,4
```

**Good**

```sql
MATCH (),()
WITH ['a','b',3.14] AS list
RETURN list,2,3,4
```

• No padding space within function call parentheses.

**Bad**

```sql
RETURN split( 'original', 'i' )
```

**Good**

```sql
RETURN split('original', 'i')
```

• Use padding space within simple subquery expressions.
A.5. Patterns

- When patterns wrap lines, break after arrows, not before.

**Bad**
```
MATCH (:Person)-->(vehicle:Car)-->(:Company)
<-(:Country)
RETURN count(vehicle)
```

**Good**
```
MATCH (:Person)-->(vehicle:Car)-->(:Company)<--
(:Country)
RETURN count(vehicle)
```

- Use anonymous nodes and relationships when the variable would not be used.

**Bad**
```
CREATE (a:End {prop: 42}),
(b:End {prop: 3}),
(c:Begin {prop: id(a)})
```

**Good**
```
CREATE (a:End {prop: 42}),
(:End {prop: 3}),
(:Begin {prop: id(a)})
```

- Chain patterns together to avoid repeating variables.

**Bad**
```
MATCH (:Person)-->(vehicle:Car), (vehicle:Car)-->(:Company)
RETURN count(vehicle)
```

**Good**
```
MATCH (:Person)-->(vehicle:Car)-->(:Company)
RETURN count(vehicle)
```

- Put named nodes before anonymous nodes.
• Keep anchor nodes at the beginning of the MATCH clause.

• Prefer outgoing (left to right) pattern relationships to incoming pattern relationships.

A.6. Meta-characters

• Use single quotes, ' ', for literal string values.

• Disregard this rule for literal strings that contain a single quote character. If the string has both, use the form that creates the fewest escapes. In the case of a tie, prefer single quotes.
Bad

RETURN 'Cypher\'s a nice language', "Mats' quote: "statement""

Good

RETURN "Cypher's a nice language", 'Mats\' quote: "statement"

- Avoid having to use back-ticks to escape characters and keywords.

Bad

MATCH (`odd-ch@racter$`:`Spaced Label` {`&property`: 42})
RETURN labels(`odd-ch@racter$`)

Good

MATCH (node:`NonSpacedLabel` {`property`: 42})
RETURN labels(node)

- Do not use a semicolon at the end of the statement.

Bad

RETURN 1;

Good

RETURN 1
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.