



# Neo4j 4.4 Cypher Manual

# Table of Contents

Overview .....	1
Tutorial .....	2
Cypher and Neo4j .....	9
Cypher and Aura .....	12
Syntax .....	14
Naming rules and recommendations .....	14
Expressions .....	15
Variables .....	21
Reserved keywords .....	21
Parameters .....	24
Operators .....	29
Comments .....	46
Patterns .....	47
Clauses .....	52
Administration clauses .....	52
Importing data .....	52
Listing functions and procedures .....	52
Multiple graphs .....	52
Projecting clauses .....	53
Reading clauses .....	53
Reading hints .....	53
Reading sub-clauses .....	53
Reading/Writing clauses .....	54
Set operations .....	54
Subquery clauses .....	54
Transaction Commands .....	55
Writing clauses .....	55
MATCH .....	55
OPTIONAL MATCH .....	68
RETURN .....	72
WITH .....	76
UNWIND .....	79
WHERE .....	83
ORDER BY .....	98
SKIP .....	102
LIMIT .....	104
CREATE .....	106
DELETE .....	111

SET .....	114
REMOVE .....	121
FOREACH .....	122
MERGE .....	123
CALL {} (subquery) .....	135
CALL procedure .....	143
UNION .....	148
USE .....	150
LOAD CSV .....	151
SHOW FUNCTIONS .....	158
SHOW PROCEDURES .....	163
Values and types .....	171
Property, structural, and composite values .....	171
Temporal values .....	173
Spatial values .....	194
Working with <code>null</code> .....	200
Lists .....	202
Maps .....	208
Casting data values .....	212
Functions .....	216
Predicate functions .....	227
Scalar functions .....	236
Aggregating functions .....	255
List functions .....	271
Mathematical functions - numeric .....	283
Mathematical functions - logarithmic .....	294
Mathematical functions - trigonometric .....	299
String functions .....	311
Temporal functions - instant types .....	323
Temporal functions - duration .....	393
Spatial functions .....	402
LOAD CSV functions .....	413
User-defined functions .....	414
Indexes for search performance .....	417
Indexes (types and limitations) .....	417
Syntax .....	418
Composite index limitations .....	421
CREATE INDEX examples .....	422
SHOW INDEXES .....	433
SHOW INDEXES examples .....	434
DROP INDEX .....	435

DROP INDEX examples .....	435
Future indexes .....	436
Deprecated syntax <a href="#">Deprecated</a> .....	446
Full-text search index .....	448
Full-text search procedures .....	449
Create and configure full-text indexes .....	450
Query full-text indexes .....	453
Drop full-text indexes .....	455
Constraints.....	457
Types of constraint .....	457
Implications on indexes .....	458
Syntax.....	458
Examples .....	463
Query tuning .....	487
Cypher query options .....	487
Profile a query.....	493
The use of indexes.....	494
Basic query tuning example.....	518
Advanced query tuning example .....	530
Planner hints and the USING keyword .....	548
Execution plans .....	567
Database hits .....	568
Execution plan operators .....	570
Execution plan operators in detail .....	580
Shortest path planning .....	698
Administration.....	704
Syntax summaries .....	704
Database management.....	705
Database alias management.....	720
Access control .....	735
Deprecations, additions and compatibility .....	839
Version 4.4 .....	839
Version 4.3 .....	846
Version 4.2 .....	852
Version 4.1.3.....	855
Version 4.1 .....	856
Version 4.0 .....	858
Version 3.5 .....	862
Version 3.4 .....	863
Version 3.3 .....	864
Version 3.2 .....	864



Version 3.1 .....	866
Version 3.0 .....	866
Compatibility .....	867
Supported language versions .....	868
Glossary of keywords .....	869
<a href="#">Clauses</a> .....	869
<a href="#">Operators</a> .....	873
<a href="#">Functions</a> .....	874
Expressions .....	881
Cypher query options .....	882
Administrative commands .....	882
Privilege Actions .....	884
Appendix A: Cypher styleguide .....	889
General recommendations .....	889
Indentation and line breaks .....	889
Casing .....	891
Spacing .....	892
Patterns .....	894
Meta-characters .....	895

# Welcome to the Neo4j Cypher Manual.

Cypher is Neo4j's declarative query language, allowing users to unlock the full potential of property graph databases.

The Cypher Manual aims to be as instructive as possible to readers from a variety of backgrounds and professions, such as developers, administrators, and academic researchers.

If you are new to Cypher and Neo4j, you can visit the [Getting Started Guide → Introduction to Cypher](#) chapter. Additionally, [Neo4j GraphAcademy](#) has a variety of free courses tailored for all levels of experience.

For a reference of all available Cypher features, see the [Cypher Cheat Sheet](#).

For a downloadable PDF version of the Cypher Manual, visit the [Neo4j documentation archive](#).

This introduction will cover the following topics:

- [Overview](#)
- [Tutorial](#)
- [Cypher and Neo4j](#)
- [Cypher and Aura](#)

License: Creative Commons 4.0

## Overview

This section provides an overview of Cypher and a brief discussion of how Cypher differs to SQL.

## What is Cypher?

Cypher is Neo4j's declarative graph query language. It was created in 2011 by Neo4j engineers as an SQL-equivalent language for graph databases. Similar to SQL, Cypher lets users focus on *what* to retrieve from graph, rather than *how* to retrieve it. As such, Cypher enables users to realize the full potential of their property graph databases by allowing for efficient and expressive queries that reveal previously unknown data connections and clusters.

Cypher provides a visual way of matching patterns and relationships. It relies on the following ascii-art type of syntax: `(nodes)-[:CONNECT_TO]->(otherNodes)`. Rounded brackets are used for circular nodes, and `-[:ARROWS]->` for relationships. Writing a query is effectively like drawing a pattern through the data in the graph. In other words, entities such as nodes and their relationships are visually built into queries. This makes Cypher a highly intuitive language to both read and write.

## Cypher and SQL: key differences

Cypher and SQL are similar in many ways. For example, they share many of the same keywords, such as `WHERE` and `ORDER BY`. However, there are some important differences between the two:

## Cypher is schema-flexible

While it is both possible and advised to enforce partial schemas using [indexes and constraints](#), Cypher and Neo4j offers a greater degree of schema-flexibility than SQL and a relational database. More specifically, nodes and relationships in a Neo4j database do not have to have a specific property set to them because other nodes or relationships in the same graph have that property (unless there is an [existence constraint created on that specific property](#)). This means that users are not required to use a fixed schema to represent data and that they can add new attributes and relationships as their graphs evolve.

## Query order

SQL queries begin with what a user wants to return, whereas Cypher queries end with the return clause. For example, consider the following two queries (both searching a database for titles of movies with a rating of greater than 7), the first written with SQL and the second with Cypher:

```
SELECT movie.name
FROM movie
WHERE movie.rating > 7
```

```
MATCH (movie:Movie)
WHERE movie.rating > 7
RETURN movie.title
```

## Cypher queries are more concise

Due to its intuitive, whiteboard-like method of constructing clauses, Cypher queries are often more concise than their equivalent SQL queries. For example, consider the following two queries (both searching a database for the names of the actors in the movie The Matrix), the first written with SQL and the second with Cypher:

```
SELECT actors.name
FROM actors
LEFT JOIN acted_in ON acted_in.actor_id = actors.id
LEFT JOIN movies ON movies.id = acted_in.movie_id
WHERE movies.title = "The Matrix"
```

```
MATCH (actor:Actor)-[:ACTED_IN]->(movie:Movie {title: 'The Matrix'})
RETURN actor.name
```

## Cypher and APOC

Neo4j supports the APOC (Awesome Procedures on Cypher) library. The APOC library provides access to user-defined procedures and functions which extend the use of the Cypher query language into areas such as data integration, graph algorithms, and data conversion.

For more details, visit the [APOC page](#).

## Tutorial

In this short tutorial, users will learn how to create, query, and delete a property graph database using Cypher. The tutorial uses the [Neo4j movie database](#).

## Creating a data model

Before creating a property graph database, it is important to develop an appropriate data model. This will provide structure to the data, and allow users of the graph to efficiently retrieve the information they are looking for.

The following data model is used for the Neo4j data model:

It includes two types of node labels:

- **Person** nodes, which have the following properties: **name** (string) and **born** (integer).
- **Movie** nodes, which have the following properties: **title** (string), **released** (integer), and **tagline** (string).

The data model also contains five different relationship types between the **Person** and **Movie** nodes: **ACTED\_IN**, **DIRECTED**, **PRODUCED**, **WROTE**, and **REVIEWED**. Two of the relationship types have properties:

- The **ACTED\_IN** relationship type, which has the **roles** property (string).
- The **REVIEWED** relationship type, which has a **summary** property (string) and a **rating** property (float).

To learn more about data modelling for graph databases, enroll in the free [Graph Data Modelling Fundamentals](#) course offered by GraphAcademy.

## Creating a property graph database

The complete Cypher query to create the Neo4j movie database, can be found [here](#). To create the full graph, run the full query against an empty Neo4j database.

## Finding nodes

The **MATCH** clause is used to find a specific pattern in the graph, such as a specific node. The **RETURN** clause specifies what of the found graph pattern to return.

For example, this query will find the nodes with **Person** label and the name **Keanu Reeves**, and return the **name** and **born** properties of the found nodes:

## Query

```
MATCH (keanu:Person {name:'Keanu Reeves'})
RETURN keanu.name, keanu.born
```

Table 1. Result

keanu.name	keanu.born
"Keanu Reeves"	1964
Rows: 1	

It is also possible to query a graph for several nodes. This query matches all nodes with the `Person` label, and limits the results to only include five rows.

## Query

```
MATCH (people:Person)
RETURN people
LIMIT 5
```

Table 2. Result

people
{"born":1964,"name":"Keanu Reeves"}
{"born":1967,"name":"Carrie-Anne Moss"}
{"born":1961,"name":"Laurence Fishburne"}
{"born":1960,"name":"Hugo Weaving"}
{"born":1967,"name":"Lilly Wachowski"}
Rows: 5

## Note on clause composition

Similar to SQL, Cypher queries are constructed using various clauses which are chained together to feed intermediate results between each other. Each clause has as input the state of the graph and a table of intermediate results consisting of the current variables. The first clause takes as input the state of the graph before the query and an empty table of intermediate results. The output of a clause is a new state of the graph and a new table of intermediate results, serving as input to the next clause. The output of the last clause is the result of the query.

Note that if one of the clauses returns an empty table of intermediate results, there is nothing to pass on to subsequent clauses, thus ending the query. (There are ways to circumvent this behaviour. For example, by replacing a `MATCH` clause with `OPTIONAL MATCH`.)

In the below example, the first `MATCH` clause finds all nodes with the `Person` label. The second clause will then filter those nodes to find all `Person` nodes who were born in the 1980s. The final clause returns the result in a descending chronological order.

## Query

```
MATCH (bornInEighties:Person)
WHERE bornInEighties.born >= 1980 AND bornInEighties.born < 1990
RETURN bornInEighties.name as name, bornInEighties.born as born ORDER BY born DESC
```

Table 3. Result

name	born
"Emile Hirsch"	1985
"Rain"	1982
"Natalie Portman"	1981
"Christina Ricci"	1980

Rows: 4

## Finding connected nodes

To discover how nodes are connected to one another, relationships must be added to queries. Queries can specify relationship types, properties, and direction, as well as the start and end nodes of the pattern.

For example, the following query matches the graph for the director of the movie the Matrix, and returns the `name` property of its directors.

## Query

```
MATCH (m:Movie {title: 'The Matrix'})<-[d:DIRECTED]-(p:Person)
RETURN p.name as director
```

Table 4. Result

director
"Lilly Wachowski"
"Lana Wachowski"

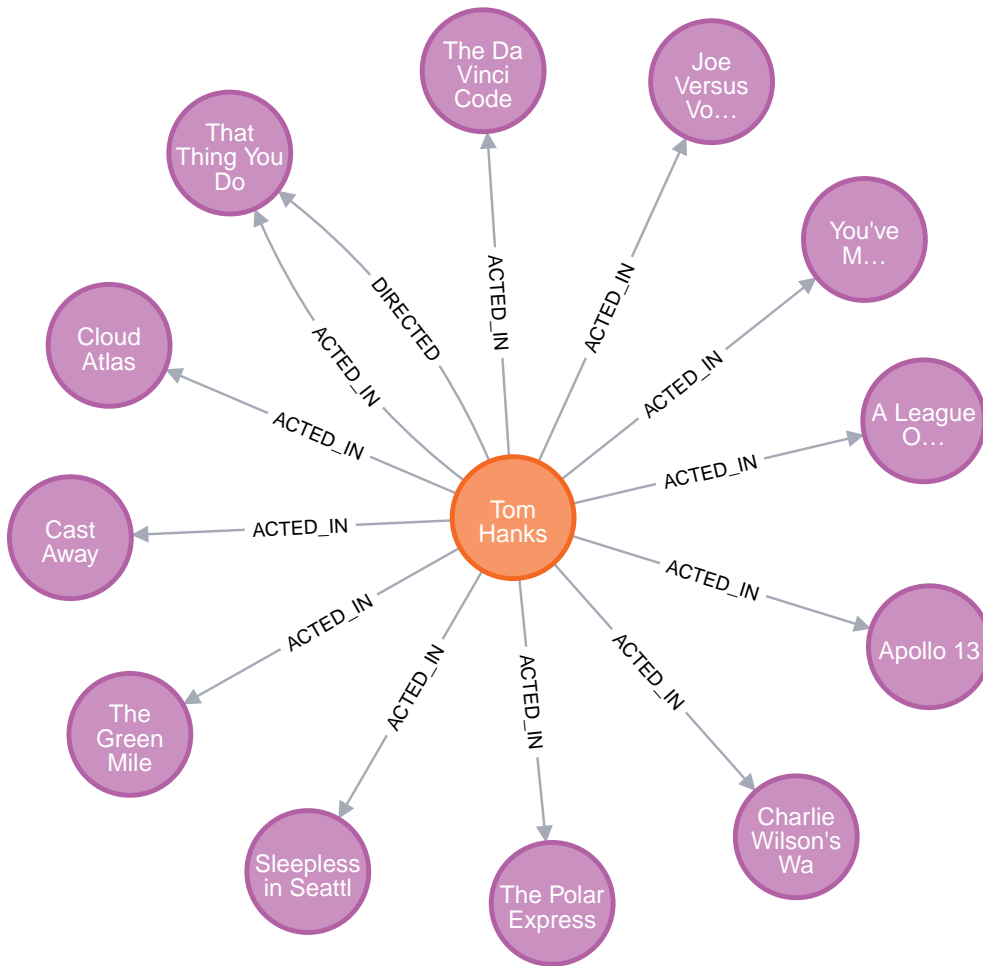
Rows: 2

It also possible to look for the type of relationships that connect nodes to one another. The below query searches the graph for outgoing relationships from the `Tom Hanks` node to any `Movie` nodes, and returns the relationships and the titles of the movies connected to him.

## Query

```
MATCH (tom:Person {name: 'Tom Hanks'})-[r]->(m:Movie)
Return tom, type(r), m.title
```

The graph returned shows that he has 13 outgoing relationships connected to 12 different `Movie` nodes (12 have the `ACTED_IN` type and one has the `DIRECTED` type).



## Finding paths

There are several ways in which Cypher can be used to search a graph for paths between nodes.

To search for patterns of a fixed length, specify the distance ("hops") between the nodes in the pattern. For example, the following query matches all **Person** nodes exactly 2 "hops" away from **Tom Hanks** and returns the first five rows.

### Query

```
MATCH (tom:Person {name:'Tom Hanks'})-[*2]-(colleagues:Person)
RETURN colleagues.name as colleagues LIMIT 5
```

Table 5. Result

colleagues
"Parker Posey"
"Greg Kinnear"
"Dave Chappelle"
"Nora Ephron"
"Meg Ryan"
Rows: 5

It is also possible to match a graph for patterns of a variable length. The below query matches all **Person** nodes up to 3 "hops" away from **Tom Hanks** and returns the first five rows. The **DISTINCT** operator ensures that the result contain no duplicate values.

Query

```
MATCH (tom:Person {name:'Tom Hanks'})-[*1..3]-(colleagues:Person)
RETURN DISTINCT colleagues.name as colleagues LIMIT 5
```

Table 6. Result

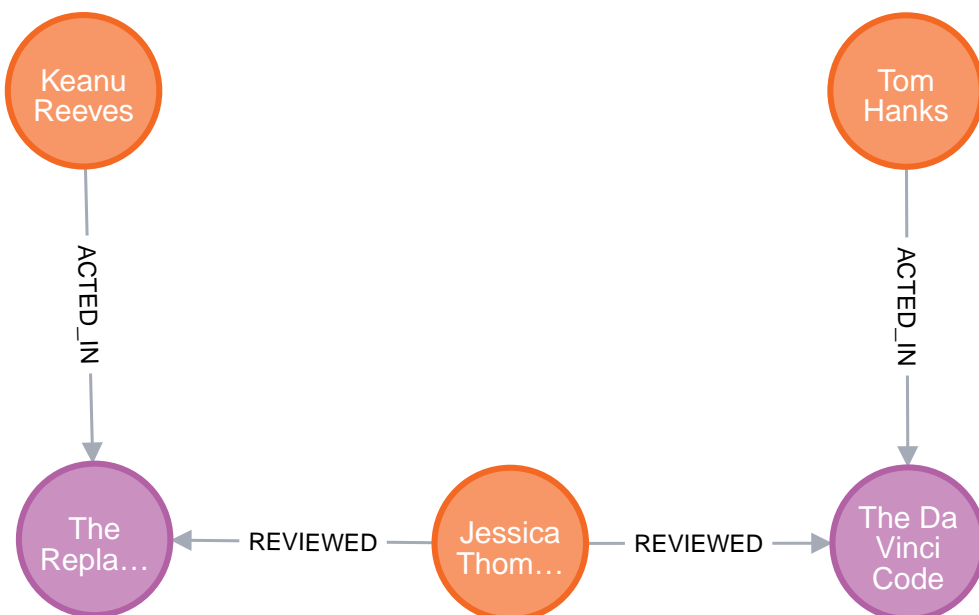
colleagues
"David Mitchell"
"Lana Wachowski"
"Halle Berry"
"Hugo Weaving"
"Jim Broadbent"
Rows: 5

To find the shortest possible path between two nodes, use the **shortestPath** algorithm. For example, this query matches the shortest path in the graph between the two nodes **Tom Hanks** and **Keanu Reeves**:

Query

```
MATCH p=shortestPath(
  (keanu:Person {name:"Keanu Reeves"})-[*]-(tom:Person {name:"Tom Hanks"})
)
RETURN p
```

This is the returned graph:



It shows that **Keanu Reeves** **ACTED\_IN** the Movie **The Replacements**, which was **REVIEWED** by the movie critic **Jessica Thompson**, who also **REVIEWED** the Movie **The Da Vinci Code** which **Tom Hanks** **ACTED\_IN**.



## Finding recommendations

Cypher allows for more complex queries. The following query tries to recommend co-actors for **Keanu Reeves**, who he has yet to work with but who his co-actors have worked with. The query then orders the results by how frequently a matched co-co-actor has collaborated with one of Keanu Reeves' co-actors.

### Query

```
MATCH (keanu:Person {name: 'Keanu Reeves'})-[:ACTED_IN]->(m:Movie)<-[:ACTED_IN]-(coActors:Person),
      (coActors:Person)-[:ACTED_IN]->(m2:Movie)<-[:ACTED_IN]-(cocoActors:Person)
WHERE NOT (keanu)-[:ACTED_IN]->()<-[:ACTED_IN]-(cocoActors) AND keanu <> cocoActors
RETURN cocoActors.name AS Recommended, count(cocoActors) AS Strength ORDER BY Strength DESC LIMIT 7
```

Table 7. Result

Recommended	Strength
"Tom Hanks"	4
"John Hurt"	3
"Jim Broadbent"	3
"Halle Berry"	3
"Stephen Rea"	3
"Natalie Portman"	3
"Ben Miles"	3

Rows: 5

There are several connections between the **Keanu Reeves** and **Tom Hanks** nodes in the movie database, but the two have never worked together in a film. The following query matches coactors who could introduce the two, by looking for co-actors who have worked with both of them in separate movies:

### Query

```
MATCH (keanu:Person {name: 'Keanu Reeves'})-[:ACTED_IN]->(m:Movie)<-[:ACTED_IN]-(coActors:Person),
      (coActors)-[:ACTED_IN]->(m2:Movie)<-[:ACTED_IN]-(tom:Person {name: 'Tom Hanks'})
RETURN DISTINCT coActors.name
```

Table 8. Result

coActors.name
"Charlize Theron"
"Hugo Weaving"

Rows: 2

## Delete a graph

To delete all nodes and relationships in a graph, run the following query:

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

# Cypher and Neo4j

This section discusses aspects of Neo4j that are important to consider when using Cypher.

## Cypher and the different editions of Neo4j

Neo4j consists of two editions: a commercial Enterprise Edition, and a Community Edition.

Cypher works almost identically between the two editions, but there are key areas in which they differ:

Feature	Enterprise Edition	Community Edition
Multi-database	Any number of user databases.	Only <b>system</b> and one user database.
Role-based security	User, role, and privilege management for flexible <b>access control</b> and <b>sub-graph access control</b> .	<b>Multi-user management</b> . All users have full access rights.
Constraints	All constraints: <b>node existence constraints</b> , <b>relationship existence constraints</b> , <b>node property uniqueness constraints</b> , <b>node key constraints</b> .	Only <b>node</b> property uniqueness constraints.

## Key Neo4j terminology

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

Refers to a data model within a database. 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.

## Built-in databases in Neo4j

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](#).

## Query considerations

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. Administrative commands execute on the `system` database. If an administrative command is submitted to a user database, it is rerouted to the system database.

## Cypher and Neo4j 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 > Memory configuration](#).

## Explicit and implicit transactions

Transactions in Neo4j can be either explicit or implicit.

Explicit	Implicit
Opened by the user.	Opened automatically.
Can execute multiple Cypher queries in sequence.	Can execute a single Cypher query.
Committed, or rolled back, by the user.	Committed automatically when a transactions finishes successfully.

Queries that start separate transactions themselves, such as queries using `CALL { ... } IN TRANSACTIONS`

or **PERIODIC COMMIT** are only allowed in *implicit* mode. Explicit transactions cannot be managed directly from queries, they must be managed via APIs or tools.

For examples of the API, or the commands used to start and commit transactions, refer to the API or tool-specific documentation:

- For information on using transactions with a Neo4j driver, see *The session API* in the [Neo4j Driver manuals](#).
- For information on using transactions over the HTTP API, see the [HTTP API documentation](#) → [Using the HTTP API](#).
- For information on using transactions within the embedded Core API, see the [Java Reference](#) → [Executing Cypher queries from Java](#).
- For information on using transactions within the Neo4j Browser or Cypher-shell, see the [Cypher-shell documentation](#).

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.

## 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.

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.

## ACID compliance

Neo4j is fully ACID compliant. This means that:

- Atomicity - If a part of a transaction fails, the database state is left unchanged.
- Consistency — Every transaction leaves the database in a consistent state.
- Isolation — During a transaction, modified data cannot be accessed by other operations.
- Durability — The DBMS can always recover the results of a committed transaction.

# Cypher and Aura

This section gives a brief overview of Aura, and how Cypher differs to users of Aura.

## What is Aura?

Aura is Neo4j's fully managed cloud service. It consists of AuraDB and AuraDS. AuraDB is a graph database service for developers building intelligent applications, and AuraDS is a Graph Data Science (GDS) service for data scientists building predictive models and analytics workflows.

AuraDB is available on the following tiers:

- AuraDB Free
- AuraDB Pro
- AuraDB Enterprise

For more information, see [Aura docs - Neo4j AuraDB overview](#).

AuraDS is available on the following tiers:

- AuraDS Pro
- AuraDS Enterprise

For more information, see [Aura docs - Neo4j AuraDS overview](#).

## Using Cypher on Aura

Most Cypher features are available on all tiers of Aura. There are, however, some features which are not available to Aura instances. For example, it is not possible to create, alter, or drop databases using Aura, nor is it possible to alter or drop servers.

There are also certain Cypher features which are only available on AuraDB Enterprise instances. These can be categorized as the role-based access-control features of Cypher. For example, it is not possible to create, alter, or drop roles using AuraDB Free, AuraDB Pro, AuraDS Pro, or AuraDS Enterprise, but it is possible using AuraDB Enterprise.

The Cypher Manual uses two different labels to differentiate this distinction:

Label	Description
Not on aura	Cypher feature not available on any tier of Aura.
Aura db enterprise	Cypher feature only available on AuraDB Enterprise.

## Aura and the Cypher Cheat Sheet

Each different tier of Aura has a customized version of the Cypher Cheat Sheet which only shows the features of Cypher available for the chosen tier.

The Cypher Cheat Sheet can be accessed [here](#). You can select your desired Aura tier and Neo4j version by using the dropdown menus provided. Note that the default tier is AuraDB Enterprise.

# Syntax

Cypher follows a several syntactical rules and recommendations that are important to know when constructing queries. Further information can be found in the following sections:

- [Naming rules and recommendations](#)
- [Expressions](#)
- [Variables](#)
- [Reserved keywords](#)
- [Parameters](#)
- [Operators](#)
- [Comments](#)
- [Patterns](#)

## Naming rules and recommendations

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

### 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`.

## Using special characters in names

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.

Within an escaped name, the following escaping sequences are allowed:

Escape sequence	Character
<code>``</code>	Backtick
<code>\uxxxx</code>	Unicode UTF-16 code point (4 hex digits must follow the <code>\u</code> )



Using escaped names with unsanitized user input makes you vulnerable to Cypher injection. Some techniques to mitigate this are:

- sanitizing (and validating) the user input.
- remodeling your data model to avoid this data access pattern.

## 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)`.

## Recommendations

Here are the recommended naming conventions:

Node labels	Camel-case, beginning with an upper-case character	<code>:VehicleOwner</code> rather than <code>:vehicle_owner</code> etc.
Relationship types	Upper-case, using underscore to separate words	<code>:OWNS_VEHICLE</code> rather than <code>:ownsVehicle</code> etc.

## Expressions

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



# 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.*'`.
- A case-sensitive string matching expression: `a.surname STARTS WITH 'Sven'`, `a.surname ENDS WITH 'son'` or `a.surname CONTAINS 'son'`.
- A `CASE` expression.

## Note on string literals

String literals can contain the following escape sequences:

Escape sequence	Character
<code>\t</code>	Tab
<code>\b</code>	Backspace
<code>\n</code>	Newline

Escape sequence	Character
<code>\r</code>	Carriage return
<code>\f</code>	Form feed
<code>\'</code>	Single quote
<code>\"</code>	Double quote
<code>\\</code>	Backslash
<code>\uxxxx</code>	Unicode UTF-16 code point (4 hex digits must follow the <code>\u</code> )



Using regular expressions with unsanitized user input makes you vulnerable to Cypher injection. Consider using [parameters](#) instead.

## Note on number literals

Any number literal may contain an underscore `_` between digits. There may be an underscore between the `0x` or `0o` and the digits for hexadecimal and octal literals.

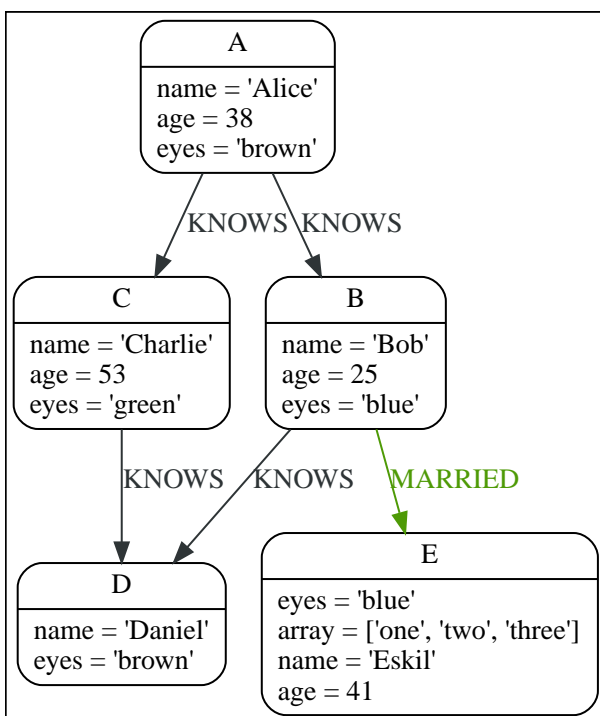
## CASE expressions

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:



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

The expression is calculated, and compared in order with the **WHEN** clauses until a match is found. If no match is found, the expression in the **ELSE** clause is returned. However, if there is no **ELSE** case and no match is found, **null** will be returned.

### Syntax:

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

### Arguments:

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

### Query

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

Table 9. Result

result
2
1
3
2
1

Rows: 5

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

The predicates are evaluated in order until a **true** value is found, and the result value is used. If no match is found, the expression in the **ELSE** clause is returned. However, if there is no **ELSE** case and no match is found, **null** will be returned.

## Syntax:

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

## Arguments:

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

## Query

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

## Table 10. Result

result
2
1
3
3
1
Rows: 5

## Distinguishing between when to use the simple and generic `CASE` forms

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

## Query

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

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

Table 11. Result

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

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

Query

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

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

Table 12. Result

n.name	age_10_years_ago
"Alice"	28
"Bob"	15
"Charlie"	43
"Daniel"	-1
"Eskil"	31
Rows: 5	

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

```
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 13. Result

(empty result)

Rows: 0

Properties set: 5

## 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:

```
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](#).

## 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.

## Clauses

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

## Subclauses

- LIMIT
- ORDER
- SKIP
- WHERE
- YIELD

## Modifiers

- ASC
- ASCENDING

- ASSERT
- BY
- CSV
- DESC
- DESCENDING
- ON

## Expressions

- ALL
- CASE
- ELSE
- END
- THEN
- WHEN

## Operators

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

## Schema

- CONSTRAINT
- CREATE
- DROP
- EXISTS
- INDEX
- NODE



- KEY
- UNIQUE

## Hints

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

## Literals

- false
- null
- true

## Reserved for future use

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

## Parameters

*This section describes parameterized querying.*

### 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.



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.

## 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.

## String literal

## Parameters

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

## Query

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

You can use parameters in this syntax as well:

## Parameters

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

## Query

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

## Regular expression

### Parameters

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

### Query

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

## Case-sensitive string pattern matching

### Parameters

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

### Query

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

## Create node with properties

### Parameters

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

### Query

```
CREATE ($props)
```

## Create multiple nodes with properties

### Parameters

```
{
  "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
```

## Setting all properties on a node

Note that this will replace all the current properties.

### Parameters

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

### Query

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

## SKIP and LIMIT

### Parameters

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

### Query

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

## Node id

### Parameters

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

### Query

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

## Multiple node ids

### Parameters

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

### Query

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

## Calling procedures

### Parameters

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

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

## 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]

## Operators at a glance

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

## Aggregation operators

The aggregation operators comprise:

- remove duplicates values: `DISTINCT`

## Using the `DISTINCT` operator

Retrieve the unique eye colors from `Person` nodes.

## Query

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

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

Table 14. Result

p.eyeColor
"blue"
"brown"

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

**DISTINCT** is commonly used in conjunction with [aggregating functions](#).

## 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 15. Result

p.name
"Jane"
"Tom"



p.name

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 16. Result

restaurant.name

"Hungry Jo"

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 17. Result

p.name	p.age	p.livesIn
"Ellen"	<null>	"London"

p.name	p.age	p.livesIn
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 18. Result

p.name	p.age	p.livesIn
"Ellen"	20	"London"
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 `+=`.

## 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 19. Result

result
8.0
Rows: 1

## Using the unary minus operator -

### Query

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

Table 20. Result

result
7
Rows: 1

## Comparison operators

The comparison operators comprise:

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

String-specific comparison operators comprise:

- STARTS WITH: perform case-sensitive prefix searching on strings
- ENDS WITH: perform case-sensitive suffix searching on strings
- CONTAINS: perform case-sensitive inclusion searching in strings
- =~: regular expression for matching a pattern

## Comparing two numbers

### Query

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

Table 21. Result

result
true
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

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

Table 22. Result

candidate
"John"
"Jonathan"
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 [Property, structural, and composite values](#)) by equality using the `=` and `<>` operators.

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

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

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

- Paths are treated as lists of alternating nodes and relationships and are equal to all lists that contain that very same sequence of nodes and relationships.
- Testing any value against `null` with both the `=` and the `<>` operators always 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 of the above is true, the points are considered incomparable and any comparison operator between them will return `null`.
- Ordering of spatial values:
  - `ORDER BY` requires all values to be orderable.
  - Points are ordered after arrays and before temporal types.
  - Points of different CRS are ordered by the CRS code (the value of SRID field). For the currently supported set of [Coordinate Reference Systems](#) this means the order: 4326, 4979, 7302, 9157
  - Points of the same CRS are ordered by each coordinate value in turn, `x` first, then `y` and finally `z`.
  - Note that this order is different to the order returned by the spatial index, which will be the order of the space filling curve.
- Comparison of temporal values:
  - [Temporal instant values](#) are comparable within the same type. An instant is considered less than another instant if it occurs before that instant in time, and it is considered greater than if it occurs after.

- Instant values that occur at the same point in time — but that have a different time zone — are not considered equal, and must therefore be ordered in some predictable way. Cypher prescribes that, after the primary order of point in time, instant values be ordered by effective time zone offset, from west (negative offset from UTC) to east (positive offset from UTC). This has the effect that times that represent the same point in time will be ordered with the time with the earliest local time first. If two instant values represent the same point in time, and have the same time zone offset, but a different named time zone (this is possible for *DateTime* only, since *Time* only has an offset), these values are not considered equal, and ordered by the time zone identifier, alphabetically, as its third ordering component. 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 <sup>[1]</sup>.
- 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
    - String

- Boolean
- Number

- 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:

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

is equivalent to

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

Thus, it 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
```

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 23. Result

word
"mouse"
"house"
Rows: 2

Further information and examples regarding the use of regular expressions in filtering can be found in [Regular expressions](#).

## Boolean operators

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

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

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

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

## Using boolean operators to filter numbers



## Query

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

Table 24. Result

number
4
7
9
Rows: 3

## String operators

The string operators comprise:

- concatenating strings: +

Concatenating two strings with +

## Query

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

Table 25. Result

result
"neo4j"
Rows: 1

## Temporal operators

Temporal operators comprise:

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

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

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

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

### Query

```
WITH
  localtime({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 26. Result

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

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

Table 27. Result

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

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

Table 28. Result

date1	date2
2012-02-28	2012-02-29
Rows: 1	

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

### Query

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

Table 29. Result

duration1	duration2	duration1 + duration2	duration1 - duration2
P12Y5M14DT16H13M10.000000000S	P1M-14DT15H49M10S	P12Y6MT32H2M20.000000001S	P12Y4M28DT24M0.000000001S
Rows: 1			

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

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

### Query

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

Table 30. Result

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

## 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 [working with null](#) page.

## 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 31. Result

p.person.name
"Anne"
Rows: 1

## 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 32. Result

result
"Anne"
Rows: 1

More information can be found in the [Maps chapter](#).

## List operators

The list operators comprise:

- concatenating lists `l1` and `l2`: `[l1] + [l2]`
- checking if an element `e` exists in a list `l`: `e 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](#).

## Concatenating two lists using `+`

Query

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

Table 33. Result

myList
[1,2,3,4,5,6,7]
Rows: 1

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

Query

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

Table 34. Result

number
2
3
Rows: 2

## 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 35. Result

inList
true
Rows: 1

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 36. Result

inList
false
Rows: 1

The following query can be used to ascertain whether or not a list — obtained from, say, the `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 `labels(n)` returns either `Person` or `Employee` (or both), the query will return a value greater than zero.

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

Query

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

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

Table 37. Result

result
["John", "Bill"]
Rows: 1

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

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

#### Query

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

Table 38. Result

result
"John"
Rows: 1

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

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

#### Query

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

Table 39. Result

result
true
Rows: 1

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

## Comments

*This section describes how how to use comments in Cypher.*

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

Examples:

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

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

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

## Patterns

*This section contains an overview of data patterns in Cypher.*

### 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](#)

### 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**.



## 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)
```

## Patterns for labels

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

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

One can also describe a node that has multiple labels:

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

## 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.

## Patterns for relationships

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

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

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

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

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

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

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

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

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

As with nodes, the name of the relationship can always be omitted, as exemplified by:

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

## Variable-length pattern matching

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

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

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

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

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

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

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

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

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

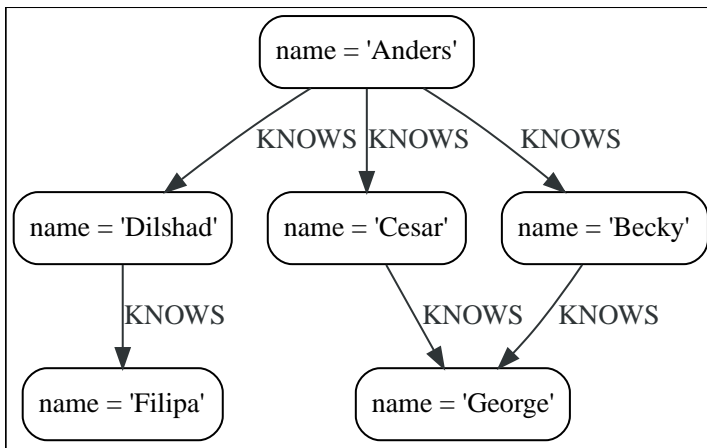
To describe paths of length 5 or less, use:

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

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

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

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



## Query

```

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

```

## Table 40. Result

remote_friend.name
"Dilshad"
"Anders"
Rows: 2

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**.

## 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.

[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  $((365 * 4 + 1) * 25 - 1) * 4 + 1 = 146097$  days, which means an average of 365.2425 days per year.

# Clauses

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

## Administration clauses

These comprise clauses used to manage databases, schema and security; further details can found in [Database management](#) and [Access control](#).

Clause	Description
<a href="#">CREATE</a>   <a href="#">DROP</a>   <a href="#">START</a>   <a href="#">STOP DATABASE</a>	Create, drop, start or stop a database.
<a href="#">CREATE</a>   <a href="#">DROP INDEX</a>	Create or drop an index on all nodes with a particular label and property.
<a href="#">CREATE</a>   <a href="#">DROP CONSTRAINT</a>	Create or drop a constraint pertaining to either a node label or relationship type, and a property.
<a href="#">Access control</a>	Manage users, roles, and privileges for database, graph and sub-graph access control.

## Importing data

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

## Listing functions and procedures

Clause	Description
<a href="#">SHOW FUNCTIONS</a>	List the available functions.
<a href="#">SHOW PROCEDURES</a>	List the available procedures.

## Multiple graphs

Clause	Description
<a href="#">USE</a>	Determines which graph a query, or query part, is executed against. <a href="#">Fabric</a>

## Projecting clauses

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

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

## Reading clauses

These comprise clauses that read data from the database.

The flow of data within a Cypher query is an unordered sequence of maps with key-value pairs — a set of possible bindings between the variables in the query and values derived from the database. This set is refined and augmented by subsequent parts of the query.

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

## Reading hints

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

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

## Reading sub-clauses

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

Sub-clause	Description
WHERE	Adds constraints to the patterns in a <b>MATCH</b> or <b>OPTIONAL MATCH</b> clause or filters the results of a <b>WITH</b> clause.
WHERE EXISTS { ... }	An existential sub-query used to filter the results of a <b>MATCH</b> , <b>OPTIONAL MATCH</b> or <b>WITH</b> clause.
<<ORDERBY, ORDER BY [ASC[ENDING]>>   DESC[ENDING]]]	A sub-clause following <b>RETURN</b> or <b>WITH</b> , specifying that the output should be sorted in either ascending (the default) or descending order.
SKIP	Defines from which row to start including the rows in the output.
LIMIT	Constrains the number of rows in the output.

## Reading/Writing clauses

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

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

## Set operations

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

## Subquery clauses

Clause	Description
CALL { ... }	Evaluates a subquery, typically used for post-union processing or aggregations.

Clause	Description
<code>CALL { ... } IN TRANSACTIONS</code>	Evaluates a subquery in separate transactions. Typically used when modifying or importing large amounts of data.

## Transaction Commands

Clause	Description
<code>SHOW TRANSACTIONS</code>	List the available transactions.
<code>TERMINATE TRANSACTIONS</code>	Terminate transactions by their IDs.

## Writing clauses

These comprise clauses that write the data to the database.

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

## 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

## 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

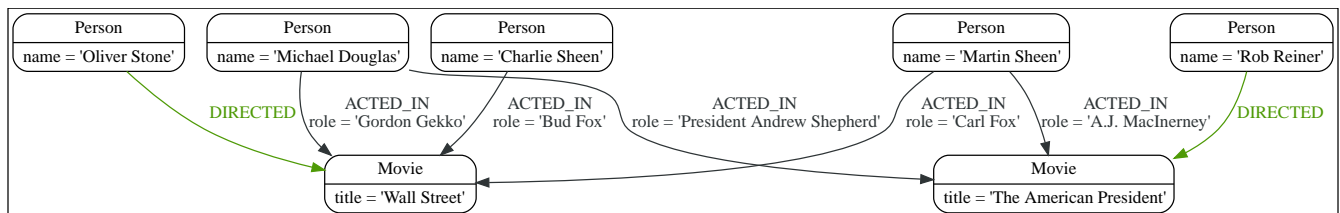
associated predicates specified in any **WHERE** part. This could involve a scan of the database, a search for nodes having a certain label, or a search of an index to find starting points for the pattern matching. Nodes and relationships found by this search are available as *bound pattern elements*, 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:



## 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 41. Result

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

## 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 42. Result

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

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 43. Result

movie.title
"Wall Street"

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 44. Result

movie.title
"Wall Street"

movie.title
Rows: 1

## 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 45. Result

movie.title
"Wall Street"
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 46. Result

type(r)
"DIRECTED"
Rows: 1

### 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 47. Result

actor.name
"Michael Douglas"
"Martin Sheen"
"Charlie Sheen"
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 48. Result

person.name
"Oliver Stone"
"Michael Douglas"
"Martin Sheen"
"Charlie Sheen"
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'})<-[r:ACTED_IN]-(actor)
RETURN r.role
```

Returns **ACTED\_IN** roles for 'Wall Street'.

Table 49. Result

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

r.role

Rows: 3

## Relationships in depth



Inside a single pattern, relationships will only be matched once.

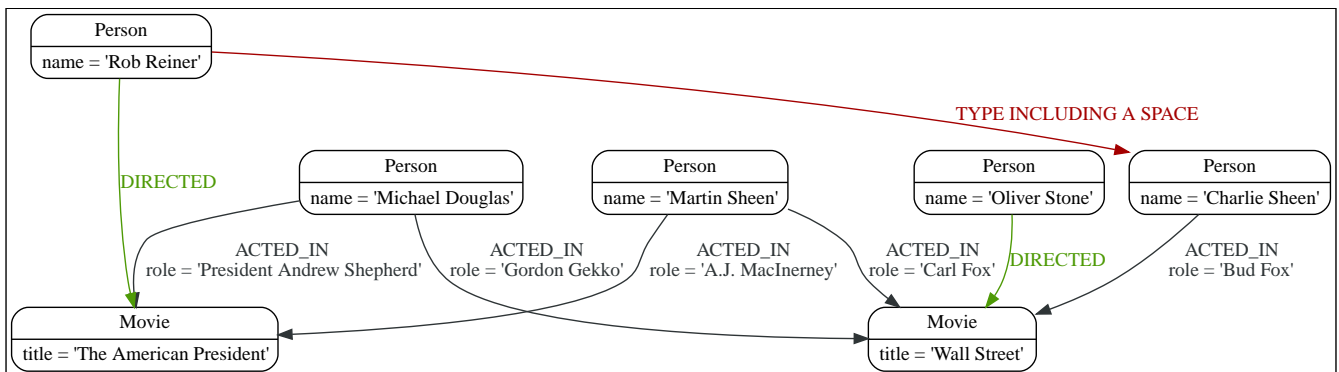
## Relationship types with uncommon characters

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:



### Query

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

Returns a relationship type with spaces in it.

Table 50. Result

type(r)
"TYPE INCLUDING A SPACE"
Rows: 1

## Multiple relationships

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

## Query

```
MATCH (charlie {name: 'Charlie Sheen'})-[:ACTED_IN]->(movie)<-[:DIRECTED]-(director)
RETURN movie.title, director.name
```

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

Table 51. Result

movie.title	director.name
"Wall Street"	"Oliver Stone"
Rows: 1	

## Variable length relationships

Nodes that are a variable number of `relationship->node` hops away can be found using the following syntax: `-[:TYPE*minHops..maxHops]->`. `minHops` and `maxHops` are optional and default to 1 and infinity respectively. When no bounds are given the dots may be omitted. The dots may also be omitted when setting only one bound and this implies a fixed length pattern.

## Query

```
MATCH (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 52. Result

movie.title
"Wall Street"
"The American President"
"The American President"
Rows: 3

## 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 53. Result

person.name
"Oliver Stone"
"Michael Douglas"
"Martin Sheen"
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 54. Result

relationships(p)
[:ACTED_IN[0]{role:"Bud Fox"},:ACTED_IN[2]{role:"Gordon Gekko"}]
[:ACTED_IN[0]{role:"Bud Fox"},:ACTED_IN[1]{role:"Carl Fox"}]
Rows: 2

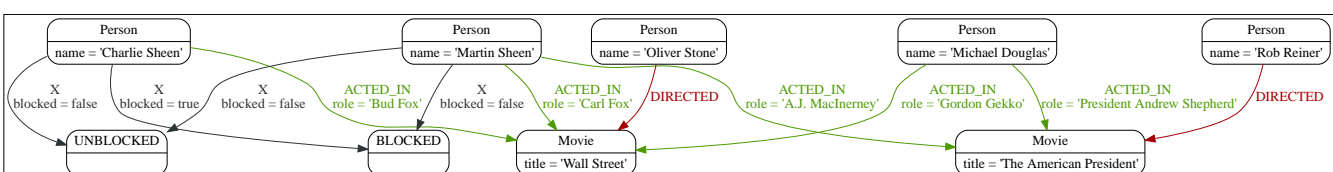
## Match with properties on a variable length path

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

Query

```
MATCH
(charlie:Person {name: 'Charlie Sheen'}),
(martin:Person {name: 'Martin Sheen'})
CREATE (charlie)-[:X {blocked: false}]>(:UNBLOCKED)<-[:X {blocked: false}]-(martin)
CREATE (charlie)-[:X {blocked: true}]>(:BLOCKED)<-[:X {blocked: false}]-(martin)
```

This means that we are starting out with the following graph:





## Query

```
MATCH p = (charlie:Person)-[* {blocked:false}]->(martin:Person)
WHERE charlie.name = 'Charlie Sheen' AND martin.name = 'Martin Sheen'
RETURN p
```

Returns the paths between 'Charlie Sheen' and 'Martin Sheen' where all relationships have the `blocked` property set to `false`.

Table 55. Result

p
<code>(0)-[X,7]-&gt;(7)&lt;-[X,8]-(1)</code>
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.

## Query

```
MATCH (wallstreet:Movie {title: 'Wall Street'})-[*0..1]->(x)
RETURN x
```

Returns the movie itself as well as actors and directors one relationship away

Table 56. Result

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

## 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 57. Result

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

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

## 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:

### Query

```
MATCH
  (martin:Person {name: 'Martin Sheen'}),
  (oliver:Person {name: 'Oliver Stone'}),
  p = shortestPath((martin)-[*..15]-(oliver))
RETURN p
```

This means: find a single shortest path between two nodes, as long as the path is max 15 relationships long. Within the parentheses you define a single link of a path — the starting node, the connecting relationship and the end node. Characteristics describing the relationship like relationship type, max hops and direction are all used when finding the shortest path. If there is a `WHERE` clause following the match of a `shortestPath`, relevant predicates will be included in the `shortestPath`. If the predicate is a `none()` or `all()` on the relationship elements of the path, it will be used during the search to improve performance (see [Shortest path planning](#)).

Table 59. Result

p
(1)-[ACTED_IN,1]->(5)<-[DIRECTED,3]-(3)
Rows: 1

## 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 60. Result

p
(0)-[ACTED_IN,0]->(5)<-[ACTED_IN,1]-(1)
Rows: 1

## 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 61. Result

p
(1)-[ACTED_IN,1]->(5)<-[ACTED_IN,2]-(2)
(1)-[ACTED_IN,4]->(6)<-[ACTED_IN,5]-(2)
Rows: 2

# Get node or relationship by ID

## Node by ID

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



Neo4j reuses its internal IDs when nodes and relationships are deleted. This means that applications using, and relying on internal Neo4j IDs, are brittle or at risk of making mistakes. It is therefore recommended to rather use application-generated IDs.

### Query

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

The corresponding node is returned.

Table 62. Result

n
<code>Node[0]{name:"Charlie Sheen"}</code>
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.

### Query

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

The relationship with ID `0` is returned.

Table 63. Result

r
<code>:ACTED_IN[0]{role:"Bud Fox"}</code>
Rows: 1

## Multiple nodes by ID

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

## Query

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

This returns the nodes listed in the `IN`-expression.

Table 64. Result

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

## OPTIONAL MATCH

### Introduction

`OPTIONAL MATCH` matches patterns against a graph database, just as `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 therefore be considered the Cypher equivalent of the outer join in SQL.

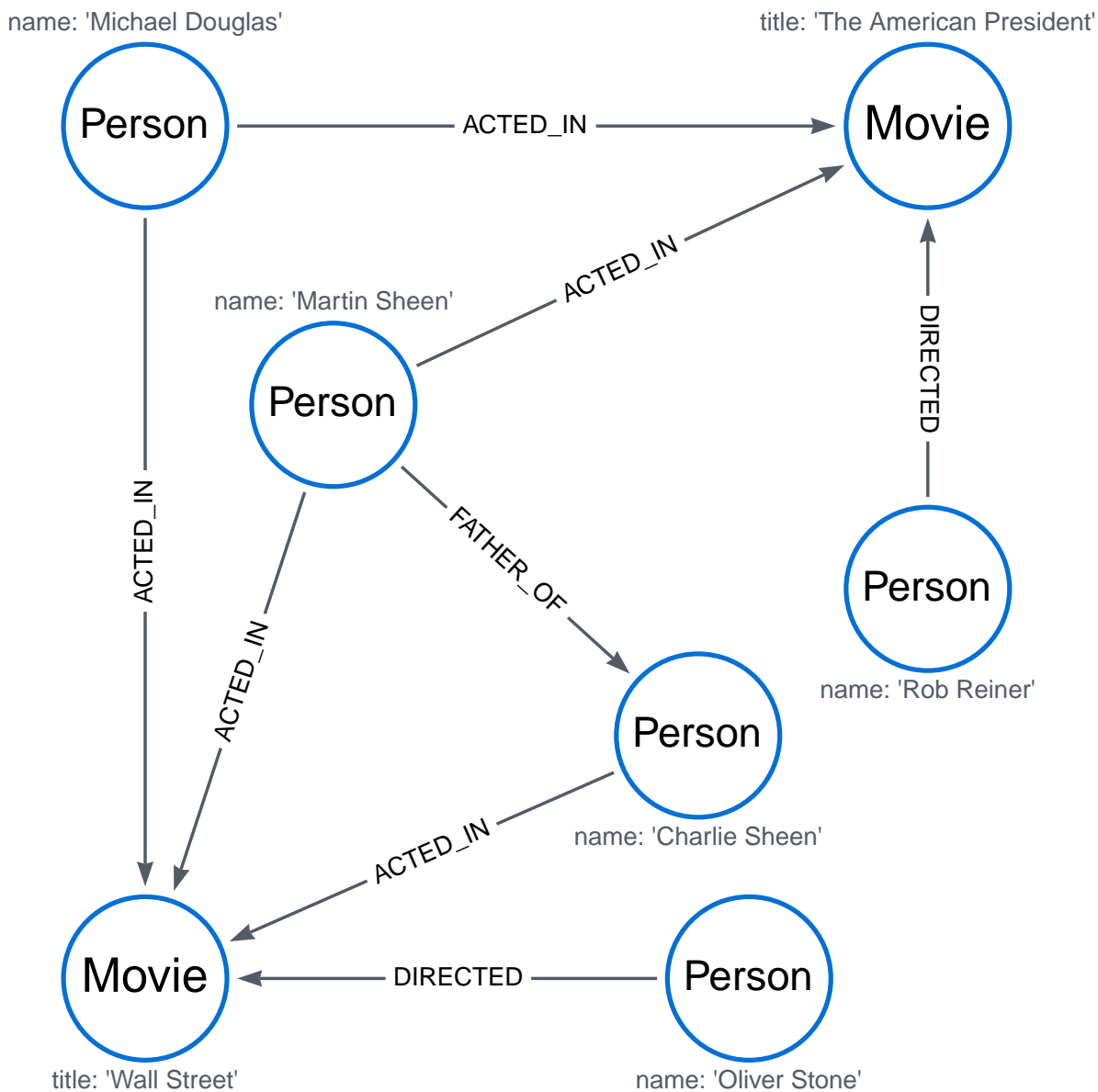
When using `OPTIONAL MATCH`, either the whole pattern is matched, or nothing is matched. The `WHERE` clause is part of the pattern description, and its predicates will be considered while looking for matches, not after. This matters especially in the case of multiple (`OPTIONAL MATCH`) clauses, where 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](#).

### Example graph

The following graph is used for the examples below:



To recreate the graph, run the following query in an empty Neo4j database:

```
CREATE
(charlie:Person {name: 'Charlie Sheen'}),
(martin:Person {name: 'Martin Sheen'}),
(michael:Person {name: 'Michael Douglas'}),
(oliver:Person {name: 'Oliver Stone'}),
(rob:Person {name: 'Rob Reiner'}),
(wallStreet:Movie {title: 'Wall Street'}),
(charlie)-[:ACTED_IN]->(wallStreet),
(martin)-[:ACTED_IN]->(wallStreet),
(michael)-[:ACTED_IN]->(wallStreet),
(oliver)-[:DIRECTED]->(wallStreet),
(thePresident:Movie {title: 'The American President'}),
(martin)-[:ACTED_IN]->(thePresident),
(michael)-[:ACTED_IN]->(thePresident),
(rob)-[:DIRECTED]->(thePresident),
(martin)-[:FATHER_OF]->(charlie)
```

## OPTIONAL MATCH in more detail

Like SQL, Cypher queries are constructed using various clauses which are chained together to feed intermediate results between each other. For example, the matching variables from one **MATCH** clause will provide the context in which the next clause exists. However, there are two important differences between

Neo4j and SQL which helps to explain **OPTIONAL MATCH** further.

1. While it is both possible and advised to enforce partial schemas using indexes and constraints, Neo4j offers a greater degree of schema flexibility than a relational database. Nodes and relationships in a Neo4j database do not have to have a specific property set to them because other nodes or relationships in the same graph have that property (unless there is an existence constraint created on the specific property).
2. Queries in Cypher are run as pipelines. If a clause returns no results, it will effectively end the query as subsequent clauses will have no data to execute upon.

For example, the following query returns no results:

```
MATCH (a:Person {name: 'Martin Sheen'})
MATCH (a)-[r:DIRECTED]->()
RETURN a.name, r
```

(no changes, no records)

This is because the second **MATCH** clause returns no data (there are no **DIRECTED** relationships connected to **Martin Sheen** in the graph) to pass on to the **RETURN** clause.

However, replacing the second **MATCH** clause with **OPTIONAL MATCH** does return results. This is because, unlike **MATCH**, **OPTIONAL MATCH** enables the value **null** to be passed between clauses.

```
MATCH (p:Person {name: 'Martin Sheen'})
OPTIONAL MATCH (p)-[r:DIRECTED]->()
RETURN p.name, r
```

Table 65. Result

p.name	r
"Martin Sheen"	<null>
Rows: 1	

**OPTIONAL MATCH** can therefore be used to check graphs for missing as well as existing values, and to pass on rows without any data to subsequent clauses in a query.

## Optional relationships

If the existence of a relationship is optional, use the **OPTIONAL MATCH** clause. If the relationship exists, it is returned. If it does not, **null** is returned in its place.

```
MATCH (a:Movie {title: 'Wall Street'})
OPTIONAL MATCH (a)-->(x)
RETURN x
```

Returns **null**, since the **Movie** node **Wall Street** has no outgoing relationships.

Table 66. Result

x
<null>
Rows: 1

On the other hand, the following query does not return `null` since the `Person` node `Charlie Sheen` has one outgoing relationship.

```
MATCH (a:Person {name: 'Charlie Sheen'})
OPTIONAL MATCH (a)-->(x)
RETURN x
```

Table 67. Result

x
{"title": "Wall Street"}
Rows: 2

## Properties on optional elements

If the existence of a property is optional, use the `OPTIONAL MATCH` clause. `null` will be returned if the specified property does not exist.

```
MATCH (a:Movie {title: 'Wall Street'})
OPTIONAL MATCH (a)-->(x)
RETURN x, x.name
```

Returns the element `x` (`null` in this query), and `null` for its `name` property, because the `Movie` node `Wall Street` has no outgoing relationships.

Table 68. Result

x	x.name
<null>	<null>
Rows: 1	

The following query only returns `null` for the nodes which lack a `name` property.

```
MATCH (a:Person {name: 'Martin Sheen'})
OPTIONAL MATCH (a)-->(x)
RETURN x, x.name
```

Table 69. Result

x	x.name
{"title": "Wall Street"}	<null>
{"name": "Charlie Sheen"}	"Charlie Sheen"
{"title": "The American President"}	<null>



x	x.name
Rows: 3	

## Optional typed and named relationship

It is also possible to look for specific relationship types when using **OPTIONAL MATCH**:

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

This returns the title of the **Movie** node **Wall Street**, and since this node has no outgoing **ACTED\_IN** relationships, **null** is returned for the relationship denoted by the variable **r**.

Table 70. Result

a.title	r
"Wall Street"	<null>
Rows: 1	

However, the following query does not return **null** since it is looking for incoming relationships of the type **ACTED\_IN** to the **Movie** node **Wall Street**.

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

Table 71. Result

a.title	x.name	type(r)
"Wall Street"	"Michael Douglas"	"ACTED_IN"
"Wall Street"	"Martin Sheen"	"ACTED_IN"
"Wall Street"	"Charlie Sheen"	"ACTED_IN"
Rows: 3		

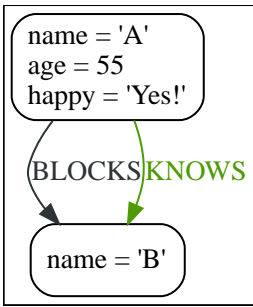
## RETURN

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

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



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



## 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 72. Result

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

## 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 73. Result

r
:KNOWS[0]{}
Rows: 1

## 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 74. Result

n.name
"A"
Rows: 1

## 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 75. Result

a	b	p	r
Node[0]{name: "A", age: 55, happy: "Yes!"}	Node[1]{name: "B"}	(0)-[BLOCKS,1]->(1)	:BLOCKS[1]{}
Node[0]{name: "A", age: 55, happy: "Yes!"}	Node[1]{name: "B"}	(0)-[KNOWS,0]->(1)	:KNOWS[0]{}
Rows: 2			

## Variable with uncommon characters

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

## Query

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

The node with name "A" is returned.

Table 76. Result

`This isn't a common variable`.happy
"Yes!"

```
`This isn't a common variable`.happy
```

Rows: 1

## 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 77. Result

SomethingTotallyDifferent
55

Rows: 1

## 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 78. Result

n.age
55
<null>

Rows: 2

## Other expressions

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

## Query

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

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

Table 79. Result

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

## Unique results

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

## Query

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

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

Table 80. Result

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

## 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.

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.

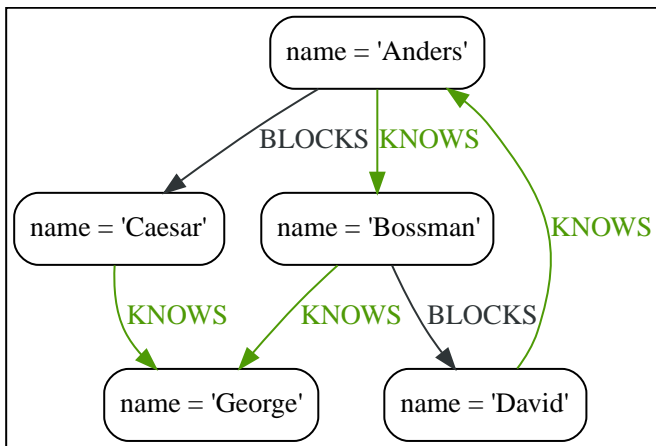
One common usage of **WITH** is to limit the number of entries passed on to other **MATCH** clauses. By combining **ORDER BY** and **LIMIT**, it is possible to get the top X entries by some criteria and then bring in additional data from the graph.

**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.



## 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 81. Result

otherPerson.name
"Caesar"

Rows: 1

## 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 82. Result

person.name	otherPerson.name	connectionType
"David"	"Anders"	"KNOWS"
"Anders"	"Bossman"	"KNOWS"
"Anders"	"Caesar"	"BLOCKS"
"Bossman"	"David"	"BLOCKS"
"Bossman"	"George"	"KNOWS"
"Caesar"	"George"	"KNOWS"

Rows: 6

## Filter on aggregate function results

Aggregated results have to pass through a **WITH** clause to be able to filter on.

## Query

```
MATCH (david {name: 'David'})--(otherPerson)-->()
WITH otherPerson, count(*) AS foaf
WHERE foaf > 1
RETURN otherPerson.name
```

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

Table 83. Result

otherPerson.name
"Anders"

Rows: 1

## 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 84. Result

collect(n.name)
["George", "David", "Caesar"]
Rows: 1

## Limit branching of a path search

You can match paths, limit to a certain number, and then match again using those paths as a base, as well as any number of similar limited searches.

### Query

```
MATCH (n {name: 'Anders'})--(m)
WITH m
ORDER BY m.name DESC
LIMIT 1
MATCH (m)--(o)
RETURN o.name
```

Starting at 'Anders', find all matching nodes, order by name descending and get the top result, then find all the nodes connected to that top result, and return their names.

Table 85. Result

o.name
"Anders"
"Bossman"
Rows: 2

## UNWIND

**UNWIND** expands a list into a sequence of rows.

The **UNWIND** clause makes it possible to transform any list back into individual rows. These lists can be parameters that were passed in, previously **collect**-ed result, or other list expressions.

Common usage of the **UNWIND** clause:

- Create distinct lists.
- Create data from parameter lists that are provided to the query.



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



## 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 86. Result

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

Rows: 4

## Creating a distinct list

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

Query

```
WITH [1, 1, 2, 2] AS coll
UNWIND coll AS x
WITH DISTINCT x
RETURN collect(x) AS setOfVals
```

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

Table 87. Result

setOfVals
[1,2]

Rows: 1

## Using `UNWIND` with any expression returning a list

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

Query

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

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

Table 88. Result

x
1
2
3
4

Rows: 4

## Using `UNWIND` with a list of lists

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

### Query

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

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

Table 89. Result

y
1
2
3
4
5

Rows: 5

## Using `UNWIND` with an empty list

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

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

## Query

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

## Table 90. Result

(empty result)

Rows: 0

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
```

## 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 91. Result

(empty result)

Rows: 0

## 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 92. Result

x
1
2

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

## 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

## Introduction

**WHERE** is not a clause in its own right — rather, it is 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.

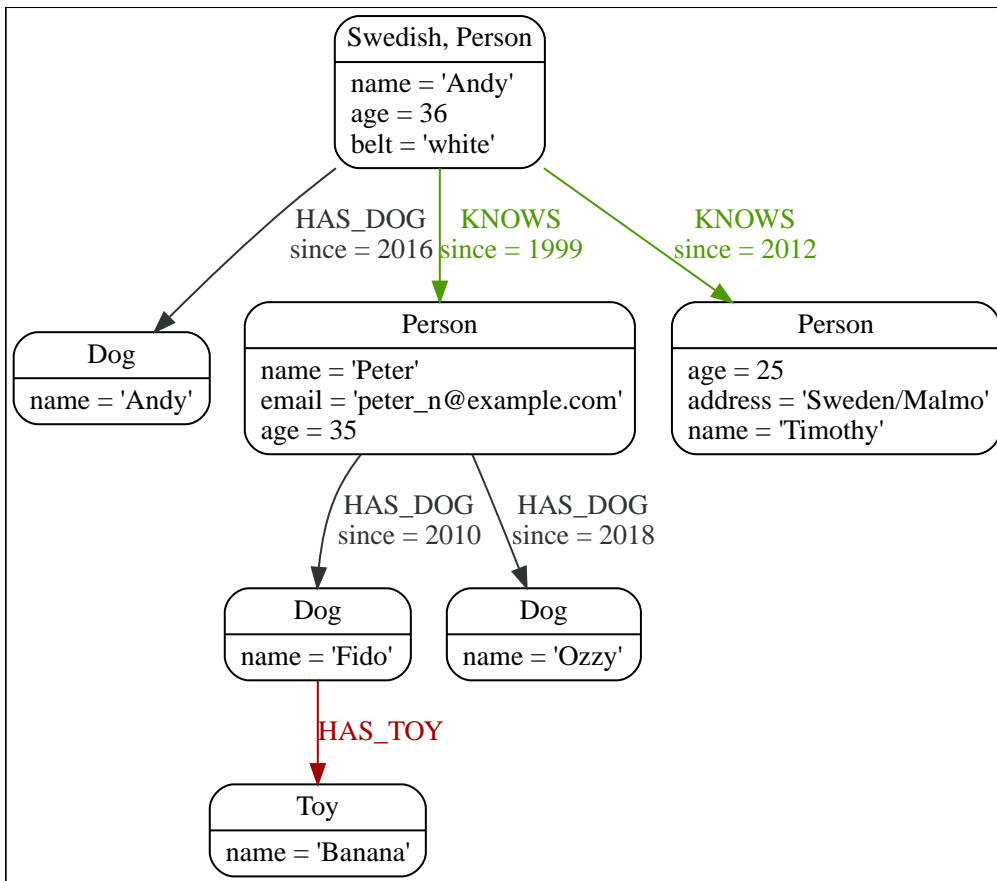


In the case of multiple **MATCH** / **OPTIONAL MATCH** clauses, the predicate in **WHERE** is always a part of the patterns in the directly preceding **MATCH** / **OPTIONAL MATCH**. Both results and performance may be impacted if the **WHERE** is put inside the wrong **MATCH** clause.



**Indexes** may be used to optimize queries using **WHERE** in a variety of cases.

The following graph is used for the examples below:



## Basic usage

## Boolean operations

The following boolean operators can be used with the **WHERE** clause: **AND**, **OR**, **XOR**, and **NOT**. For more information on how operators work with **null**, see the chapter on [Working 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 93. Result

name	age
"Andy"	36
"Peter"	35
"Timothy"	25

Rows: 3

## 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 94. Result

n.name	n.age
"Andy"	36
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 95. Result

n.name	n.age
"Timothy"	25
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 96. Result

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

## Filter on dynamically-computed node property

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

### Query

```
WITH 'AGE' AS propname
MATCH (n:Person)
WHERE n[toLower(propname)] < 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 97. Result

n.name	n.age
"Timothy"	25
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 98. Result

n.name	n.belt
"Andy"	"white"
Rows: 1	

## 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 99. Result

n.name	n.age
"Peter"	35

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 ENDS WITH 'ter'
RETURN n.name, n.age
```

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

Table 100. Result

n.name	n.age
"Peter"	35

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 101. Result

n.name	n.age
"Peter"	35

n.name	n.age
Rows: 1	

## String matching negation

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

Query

```
MATCH (n:Person)
WHERE NOT n.name ENDS WITH 'y'
RETURN n.name, n.age
```

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

Table 102. Result

n.name	n.age
"Peter"	35
Rows: 1	

## 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:

Query

```
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 103. Result

n.name	n.age
"Timothy"	25
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 104. Result

n.name	n.age	n.email
"Peter"	35	"peter_n@example.com"
Rows: 1		

Note that the regular expression constructs in [Java regular expressions](#) are applied only after resolving the escaped character sequences in the given [string literal](#). It is sometimes necessary to add additional backslashes to express regular expression constructs. This list clarifies the combination of these two definitions, containing the original escape sequence and the resulting character in the regular expression:

String literal sequence	Resulting Regex sequence	Regex match
<code>\t</code>	Tab	Tab
<code>\\t</code>	<code>\\t</code>	Tab
<code>\b</code>	Backspace	Backspace
<code>\\b</code>	<code>\\b</code>	Word boundary
<code>\n</code>	Newline	NewLine
<code>\\n</code>	<code>\\n</code>	Newline
<code>\r</code>	Carriage return	Carriage return
<code>\\r</code>	<code>\\r</code>	Carriage return
<code>\f</code>	Form feed	Form feed
<code>\\f</code>	<code>\\f</code>	Form feed
<code>\'</code>	Single quote	Single quote
<code>\"</code>	Double quote	Double quote
<code>\\</code>	Backslash	Backslash

String literal sequence	Resulting Regex sequence	Regex match
<code>\\</code>	<code>\\</code>	Backslash
<code>\uxxxx</code>	Unicode UTF-16 code point (4 hex digits must follow the <code>\u</code> )	Unicode UTF-16 code point (4 hex digits must follow the <code>\u</code> )
<code>\\uxxxx</code>	<code>\\uxxxx</code>	Unicode UTF-16 code point (4 hex digits must follow the <code>\u</code> )



Using regular expressions with unsanitized user input makes you vulnerable to Cypher injection. Consider using [parameters](#) instead.

## Case-insensitive regular expressions

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

### Query

```
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 105. Result

n.name	n.age
"Andy"	36
Rows: 1	

## 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 list 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

```
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 106. Result

other.name	other.age
"Andy"	36
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 107. Result

person.name	person.age
"Timothy"	25
"Peter"	35
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 relationship with the KNOWS-type, to a node with the property-key name and value 'Timothy'.

Table 108. Result

n.name	n.age
"Andy"	36
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 109. Result

type(r)	r.since
"KNOWS"	1999
"KNOWS"	2012
Rows: 2	

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.

## 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

```
MATCH (person:Person)
WHERE EXISTS {
  MATCH (person)-[:HAS_DOG]->(Dog)
}
RETURN person.name AS name
```

Table 110. Result

name
"Andy"
"Peter"
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

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

Table 111. Result

name
"Andy"
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

```
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 112. Result

name
"Peter"
Rows: 1

## Lists

### IN operator

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

## Query

```
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 113. Result

a.name	a.age
"Timothy"	25
"Peter"	35
Rows: 2	

## Missing properties and values

### Default to **false** if property is missing

As missing properties evaluate to **null**, the comparison in the example will evaluate to **false** for nodes without the **belt** property.

## Query

```
MATCH (n: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 114. Result

n.name	n.age	n.belt
"Andy"	36	"white"

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

```
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 115. Result

n.name	n.age	n.belt
"Andy"	36	"white"
"Peter"	35	<null>
"Timothy"	25	<null>

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

```
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 116. Result

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

Rows: 1

## 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 117. Result

a.name	a.age
"Timothy"	25
"Peter"	35
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 118. Result

a.name	a.age
"Peter"	35
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 119. Result

b.name
"Peter"
Rows: 1

When used this way, predicates in **WHERE** can reference the node variable that the **WHERE** clause belongs to, but not other elements of the **MATCH** pattern.

The same rule applies to pattern comprehensions:

## Query

```
MATCH (a:Person {name: 'Andy'})
RETURN [(a)-->(b WHERE b:Person) | b.name] AS friends
```

Table 120. Result

friends
["Peter", "Timothy"]
Rows: 1

# ORDER BY

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

**ORDER BY** relies on comparisons to sort the output, see [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 **ORDER BY**.

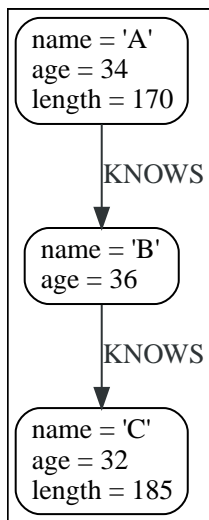
In terms of scope of variables, **ORDER BY** follows special rules, depending on if the projecting **RETURN** or **WITH** clause is either aggregating or **DISTINCT**. If it is an aggregating or **DISTINCT** projection, only the variables available in the projection are available. If the projection does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

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

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 [Index-backed ORDER BY](#).

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](#).

## 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 121. Result

n.name	n.age
"A"	34
"B"	36
"C"	32

Rows: 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 122. Result

n.name	n.age
"C"	32
"A"	34
"B"	36

Rows: 3

## 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 123. Result

n.name	n.age
"A"	34
"B"	36
"C"	32

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.

## 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 124. Result

n.name	n.age	n.length
"B"	36	<null>
"A"	34	170
"C"	32	185

Rows: 3

## 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 125. Result

n.name	n.age
"C"	32
"B"	36
"A"	34

Rows: 3

## 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 126. Result

n.length	n.name	n.age
170	"A"	34
185	"C"	32
<null>	"B"	36

Rows: 3

## 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 127. Result

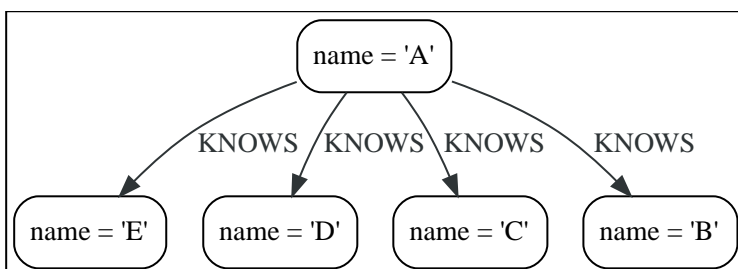
names
["C", "A", "B"]

Rows: 1

## SKIP

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

By using **SKIP**, the result set will get trimmed from the top. Please note that no guarantees are made on the order of the result unless the query specifies the **ORDER BY** clause. **SKIP** accepts any expression that evaluates to a positive integer — however the expression cannot refer to nodes or relationships.



## Skip first three rows

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

Query

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

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

Table 128. Result

n.name
"D"
"E"
Rows: 2

## Return middle two rows

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

Query

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

Two nodes from the middle are returned.

Table 129. Result

n.name
"B"
"C"
Rows: 2

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

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

Query

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



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

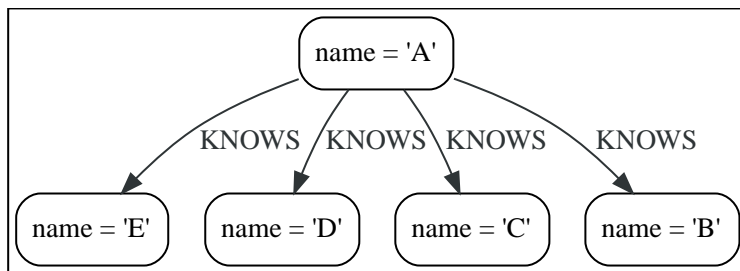
Table 130. Result

n.name
"B"
"C"
"D"
"E"
Rows: 4

## LIMIT

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

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



## Return a limited subset of the rows

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

Query

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

Limit to 3 rows by the example query.

Table 131. Result

n.name
"A"
"B"
"C"
Rows: 3

## Using an expression with **LIMIT** to return a subset of the rows

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

Query

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

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

Table 132. Result

n.name
"A"
"B"
"C"
Rows: 3

## **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 Neo4j versions before 4.3.

Query

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

This query returns nothing, but creates one node:

Table 133. 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 134. 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 135. Result

n

Node[0]{locked:true,name:"A"}

Rows: 1

Properties set: 1

## 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.

## Create nodes

### Create single node

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

Query

```
CREATE (n)
```

Table 136. Result

(empty result)

Rows: 0

Nodes created: 1

### Create multiple nodes

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

Query

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

Table 137. Result

(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 138. Result

(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 139. Result

(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 140. Result

(empty result)

Rows: 0  
Nodes created: 1  
Properties set: 2  
Labels added: 1

## Return created node

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

## Query

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

The name of the newly-created node is returned.

## Table 141. Result

a.name

"Andy"

Rows: 1  
Nodes created: 1  
Properties set: 1

## Create relationships

## Create a relationship between two nodes

To create a relationship between two nodes, we first get the two nodes. Once the nodes are loaded, we simply create a relationship between them.

### Query

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

The created relationship is returned by the query.

Table 142. Result

type(r)
"RELTYPE"
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 143. Result

type(r)	r.name
"RELTYPE"	"A<->B"
Rows: 1 Relationships created: 1 Properties set: 1	

## Create a full path

When you use **CREATE** and a pattern, all parts of the pattern that are not already in scope at this time will be created.

## Query

```
CREATE p = (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 144. Result

p
(2)-[:WORKS_AT,0]->(3)<-[:WORKS_AT,1]-(4)
Rows: 1 Nodes created: 3 Relationships created: 2 Properties set: 2

## Use parameters with CREATE

### Create node with a parameter for the properties

You can also create a graph entity from a map. All the key/value pairs in the map will be set as properties on the created relationship or node. In this case we add a `Person` label to the node as well.

#### Parameters

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

## Query

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

Table 145. Result

n
Node[2]{name:"Andy",position:"Developer"}
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

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

## Query

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

Table 146. Result

(empty result)

Rows: 0

Nodes created: 2

Properties set: 4

## DELETE

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

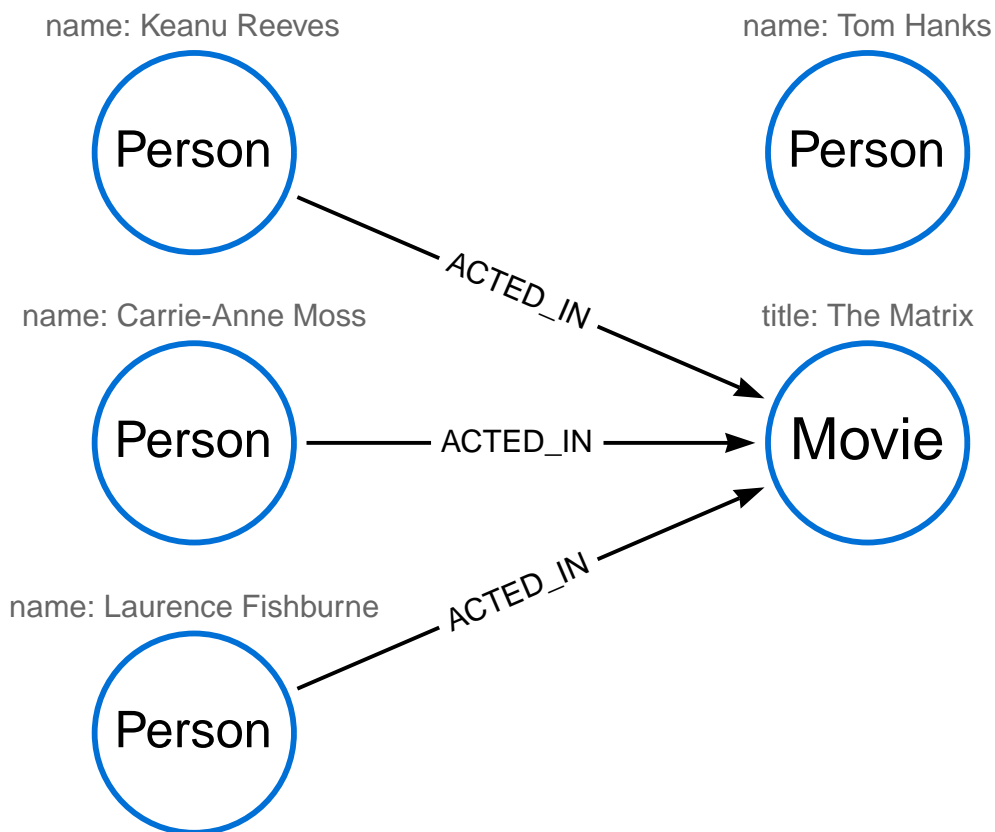
For removing properties and labels, see the **REMOVE** clause.

It is not possible to delete nodes with relationships connected to them without also deleting the relationships. This can be done by either explicitly deleting specific relationships, or by using the **DETACH DELETE** clause.

## Example graph

The following graph is used for the examples below. It shows four actors, three of whom **ACTED\_IN** the **Movie The Matrix** (Keanu Reeves, Carrie-Anne Moss, and Laurence Fishburne), and one actor who did not act in it (Tom Hanks).





To recreate the graph, run the following query in an empty Neo4j database:

```

CREATE
  (keanu:Person {name: 'Keanu Reever'}),
  (laurence:Person {name: 'Laurence Fishburne'}),
  (carrie:Person {name: 'Carrie-Anne Moss'}),
  (tom:Person {name: 'Tom Hanks'}),
  (theMatrix:Movie {title: 'The Matrix'}),
  (keanu)-[:ACTED_IN]->(theMatrix),
  (laurence)-[:ACTED_IN]->(theMatrix),
  (carrie)-[:ACTED_IN]->(theMatrix)
  
```

## Delete single node

To delete a single node, use the **DELETE** clause:

Query

```

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

This deletes the **Person** node **Tom Hanks**. This query is only possible to run on nodes without any relationships connected to them.

Result

```

Deleted 1 node
  
```

## Delete relationships only

It is possible to delete a relationship while leaving the node(s) connected to that relationship otherwise unaffected.

Query

```
MATCH (n:Person {name: 'Laurence Fishburne'})-[r:ACTED_IN]->()
DELETE r
```

This deletes all outgoing `ACTED_IN` relationships from the `Person` node `Laurence Fishburne`, without deleting the node.

Result

```
Deleted 1 relationship
```

## Delete a node with all its relationships

To delete nodes and any relationships connected them, use the `DETACH DELETE` clause.

Query

```
MATCH (n:Person {name: 'Carrie-Anne Moss'})
DETACH DELETE n
```

This deletes the `Person` node `Carrie-Anne Moss` and all relationships connected to it.

Result

```
Deleted 1 node, deleted 1 relationship
```



The `DETACH DELETE` clause may not be permitted to users with restricted security privileges. For more information, see [Operations Manual](#) → [Fine-grained access control](#).

## Delete all nodes and relationships

It is possible to delete all nodes and relationships in a graph.

Query

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

Result

```
Deleted 3 nodes, deleted 1 relationship
```



This query is not for deleting large amounts of data, but is useful when experimenting with small example datasets. When deleting large amounts of data, instead use `CALL { ... } IN TRANSACTIONS`.

## SET

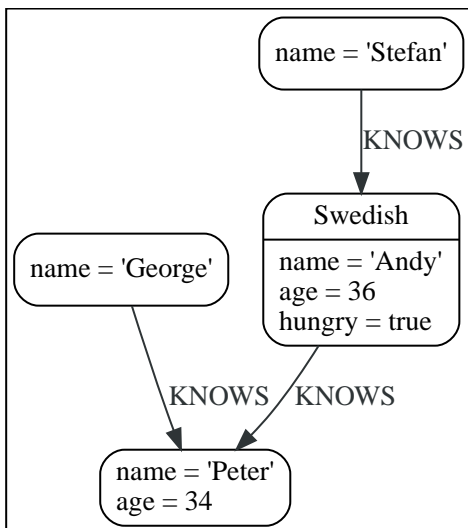
The `SET` clause is used to update labels on nodes and properties on nodes and relationships.

`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:



### Set a property

Update a node property:

Query

```

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

The newly-changed node is returned by the query.

Table 147. Result

n.name	n.surname
"Andy"	"Taylor"

n.name	n.surname
Rows: 1	
Properties set: 1	

Update a relationship property:

Query

```
MATCH (n:Swedish {name: 'Andy'})-[:KNOWS]->(m)
SET r.since = 1999
RETURN r, m.name AS friend
```

Table 148. Result

r	friend
<code>[:KNOWS {since: 1999}]</code>	"Peter"
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 149. Result

n.name	n.worksIn
"Andy"	"Malmo"
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 150. Result

n.name	n.worksIn
"Andy"	<null>

n.name	n.worksIn
Rows: 1	

## 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 151. Result

n.name	n.age
"Andy"	"36"
Rows: 1	Properties set: 1

## Remove a property

Although **REMOVE** is normally used to remove a property, it is 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 152. Result

n.name	n.age
<null>	36
Rows: 1	Properties set: 1

## 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
```

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

Table 153. Result

at.name	at.age	at.hungry	pn.name	pn.age
"Peter"	34	<null>	"Peter"	34

Rows: 1  
Properties set: 3

## 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 154. Result

p.name	p.age	p.position
"Peter Smith"	<null>	"Entrepreneur"

Rows: 1  
Properties set: 3

## 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 155. Result

p.name	p.age
<null>	<null>

Rows: 1  
Properties set: 2

## 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

```
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 156. Result

p.name	p.age	p.hungry	p.position
"Peter"	38	true	"Entrepreneur"

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

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

Table 157. Result

p.name	p.age
"Peter"	34

Rows: 1

## Set multiple properties using one SET clause

Set multiple properties at once by separating them with a comma:

Query

```
MATCH (n {name: 'Andy'})
SET n.position = 'Developer', n.surname = 'Taylor'
```

Table 158. Result

(empty result)

Rows: 0

Properties set: 2

## Set a property using a parameter

Use a parameter to set the value of a property:

Parameters

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

Query

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

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

Table 159. Result

n.name	n.surname
"Andy"	"Taylor"

Rows: 1  
Properties set: 1

## Set all properties using a parameter

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

Parameters

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



## Query

```
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 160. Result

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

Rows: 1  
Properties set: 4

## 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 161. Result

n.name	labels
"Stefan"	["German"]

Rows: 1  
Labels added: 1

## 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 162. Result

n.name	labels
"George"	["Swedish", "Bossman"]

n.name	labels
Rows: 1	
Labels added: 2	

## REMOVE

The **REMOVE** clause is used to remove properties from nodes and relationships, and to remove labels from nodes.

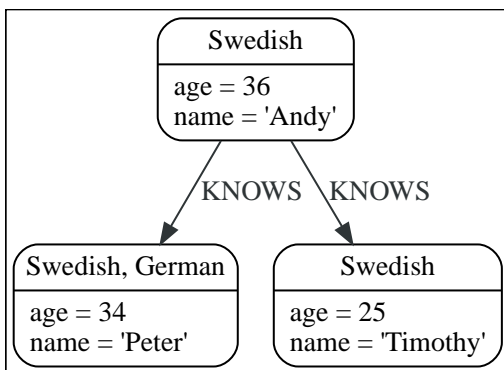


For deleting nodes and relationships, see **DELETE**.



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:



## 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 163. Result

a.name	a.age
"Andy"	<null>
Rows: 1	
Properties set: 1	

## 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.

## Remove a label from a node

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

### Query

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

Table 164. Result

n.name	labels(n)
"Peter"	["Swedish"]

Rows: 1  
Labels removed: 1

## Remove multiple labels from a node

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

### Query

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

Table 165. Result

n.name	labels(n)
"Peter"	[]

Rows: 1  
Labels removed: 2

## FOREACH

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

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.

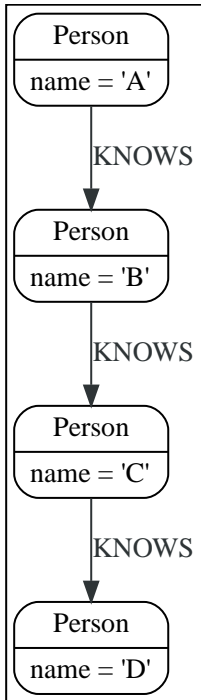
The variable context within the **FOREACH** parenthesis is separate from the one outside it. This means that if

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.



## 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 166. Result

(empty result)

Rows: 0

Properties set: 4

## 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**

## 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.

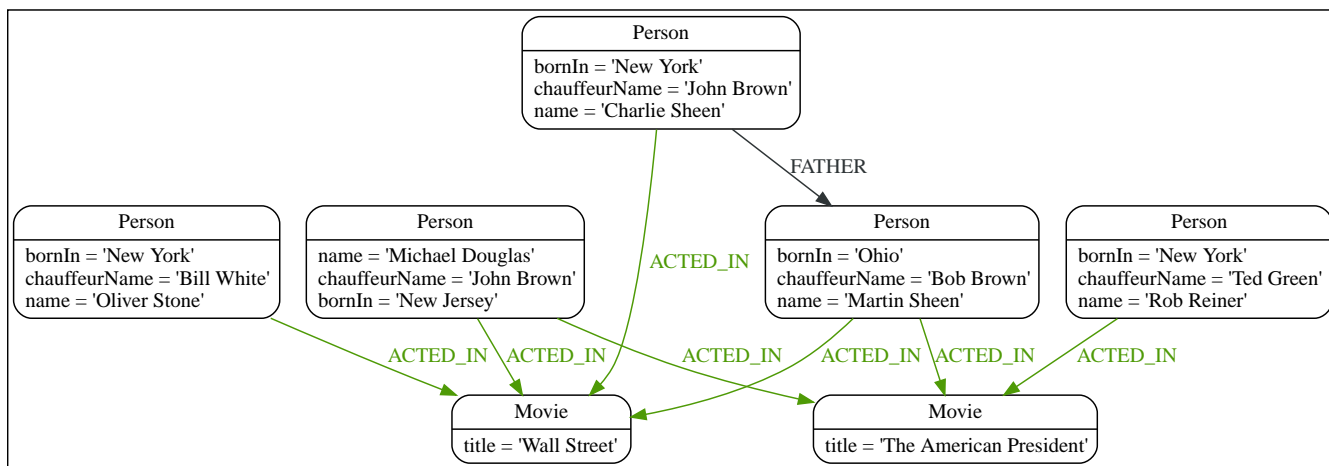


Under concurrent updates, **MERGE** only guarantees existence of the **MERGE** pattern, but not uniqueness. To guarantee uniqueness of nodes with certain properties, a **unique constraint** should be used. See [Using unique constraints with MERGE](#) to see how **MERGE** can be used in combination with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



## 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 167. Result

robert	labels(robert)
Node[7]{}	["Critic"]
Rows: 1	
Nodes created: 1	
Labels added: 1	

## Merge single node with properties

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

### Query

```
MERGE (charlie {name: 'Charlie Sheen', age: 10})
RETURN charlie
```

A new node with the name **Charlie Sheen** is created since not all properties matched those set to the pre-existing **Charlie Sheen** node:

Table 168. Result

charlie
Node[7]{age:10,name:"Charlie Sheen"}
Rows: 1 Nodes created: 1 Properties set: 2

**MERGE** cannot be used for nodes with property values that are **null**. For example, the following query will throw an error:

### Query

```
MERGE (martin:Person {name: 'Martin Sheen', age: null})
RETURN martin
```

Cannot merge the following node because of null property value for 'age': (:Person {age: null})

## Merge single node specifying both label and property

Merging a single node with both label and property matching an existing node.

### Query

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

**Michael Douglas** is matched and the **name** and **bornIn** properties are returned:

Table 169. Result

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

## Query

```
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

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

In the above query, three nodes labeled `Location` 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 `Location` 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 170. Result

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

Rows: 5  
Nodes created: 3  
Properties set: 3  
Labels added: 3

## Use `ON CREATE` and `ON MATCH`

### Merge with `ON CREATE`

Merge a node and set properties if the node needs to be created.

## Query

```
MERGE (keanu:Person {name: 'Keanu Reeves'})
ON CREATE
  SET keanu.created = timestamp()
RETURN keanu.name, keanu.created
```

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



Table 171. Result

keanu.name	keanu.created
"Keanu Reeves"	1655200898563

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

## Merge with ON MATCH

Merging nodes and setting properties on found nodes.

### Query

```
MERGE (person:Person)
ON MATCH
  SET person.found = true
RETURN person.name, person.found
```

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

Table 172. Result

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

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 173. Result

keanu.name	keanu.created	keanu.lastSeen
"Keanu Reeves"	1655200902354	<null>

keanu.name	keanu.created	keanu.lastSeen
Rows: 1 Nodes created: 1 Properties set: 2 Labels added: 1		

## Merge with **ON MATCH** setting multiple properties

If multiple properties should be set, simply separate them with commas.

Query

```
MERGE (person:Person)
ON MATCH
SET
  person.found = true,
  person.lastAccessed = timestamp()
RETURN person.name, person.found, person.lastAccessed
```

Table 174. Result

person.name	person.found	person.lastAccessed
"Charlie Sheen"	true	1655200903558
"Martin Sheen"	true	1655200903558
"Michael Douglas"	true	1655200903558
"Oliver Stone"	true	1655200903558
"Rob Reiner"	true	1655200903558
Rows: 5 Properties set: 10		

## Merge relationships

### Merge on a relationship

**MERGE** can be used to match or create a relationship.

Query

```
MATCH
  (charlie:Person {name: 'Charlie Sheen'}),
  (wallStreet:Movie {title: 'Wall Street'})
MERGE (charlie)-[r:ACTED_IN]->(wallStreet)
RETURN charlie.name, type(r), wallStreet.title
```

**Charlie Sheen** had already been marked as acting in **Wall Street**, so the existing relationship is found and returned. Note that in order to match or create a relationship when using **MERGE**, at least one bound node must be specified, which is done via the **MATCH** clause in the above example.

Table 175. Result

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

Rows: 1

**MERGE** cannot be used for relationships with property values that are `null`. For example, the following query will throw an error:

Query

```
MERGE (martin:Person {name: 'Martin Sheen'})-[r:FATHER_OF {since: null}]->(charlie:Person {name: 'Charlie Sheen'})
RETURN type(r)
```

Cannot merge the following relationship because of null property value for 'since':  
(martin)-[:FATHER\_OF {since: null}]->(charlie)

## Merge on multiple relationships

Query

```
MATCH
  (oliver:Person {name: 'Oliver Stone'}),
  (reiner:Person {name: 'Rob Reiner'})
MERGE (oliver)-[:DIRECTED]->(movie:Movie)-[:ACTED_IN]-(reiner)
RETURN movie
```

In the example graph, `Oliver Stone` and `Rob Reiner` have never worked together. When trying to `MERGE` a `Movie` node between them, Neo4j will not use any of the existing `Movie` nodes already connected to either person. Instead, a new `Movie` node is created.

Table 176. Result

movie
Node[7]{}

Rows: 1  
Nodes created: 1  
Relationships created: 2  
Labels added: 1

## Merge on an undirected relationship

`MERGE` can also be used with an undirected relationship. When it needs to create a new one, it will pick a direction.

Query

```
MATCH
  (charlie:Person {name: 'Charlie Sheen'}),
  (oliver:Person {name: 'Oliver Stone'})
MERGE (charlie)-[:KNOWS]-(oliver)
RETURN r
```

As **Charlie Sheen** and **Oliver Stone** do not know each other in the example graph, this **MERGE** query will create a **KNOWS** relationship between them. The direction of the created relationship is left to right.

Table 177. Result

r
:KNOWS[8]{}
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 location 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 **Location** node (**New York**).

Table 178. Result

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

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

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

## Using unique constraints with **MERGE**

Cypher prevents getting conflicting results from **MERGE** when using patterns that involve unique constraints. In this case, there must be at most one node that matches that pattern.

For example, given two unique constraints on **:Person(id)** and **:Person(ssn)**, a query such as **MERGE (n:Person {id: 12, ssn: 437})** will fail, if there are two different nodes (one with **id** 12 and one with **ssn** 437) or if there is only one node with only one of the properties. In other words, there must be exactly one node that matches the pattern, or no matching nodes.

Note that the following examples assume the existence of unique constraints that have been created using:

```
CREATE CONSTRAINT 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

```
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 180. Result

laurence.name
"Laurence Fishburne"

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

## Merge using unique constraints matches an existing node

Merge using unique constraints matches an existing node.

### Query

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

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

### Table 181. Result

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

Rows: 1

## Merge with unique constraints and partial matches

Merge using unique constraints fails when finding partial matches.

### Query

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

While there is a matching unique Person node with the name Michael Douglas, there is no unique node with the role of Gordon Gekko and MERGE, therefore, 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
```

To set the **role** of **Gordon Gekko** to **Michael Douglas**, use the **SET** clause instead:

## Query

```
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

```
MERGE (oliver:Person {name: 'Oliver Stone', role: 'Gordon Gekko'})
RETURN oliver
```

While there is a matching unique **Person** node with the name **Oliver Stone**, there is also another unique **Person** 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

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

Table 182. Result

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

person.name	person.role
Rows: 1	
Nodes created: 1	
Properties set: 2	
Labels added: 1	

## CALL {} (subquery)

The **CALL {}** clause evaluates a subquery that returns some values.

**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.



The **CALL** clause is also used for calling procedures. For descriptions of the **CALL** clause in this context, refer to **CALL procedure**.

Subqueries which end in a **RETURN** statement are called *returning subqueries* while subqueries without such a return statement are called *unit subqueries*.

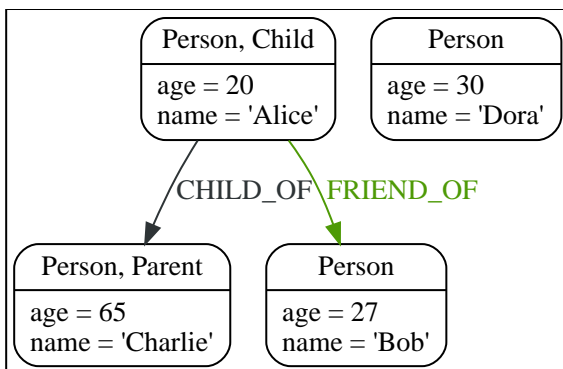
A subquery is evaluated for each incoming input row. Every output row of a *returning subquery* is combined with the input row to build the result of the subquery. That means that a returning subquery will influence the number of rows. If the subquery does not return any rows, there will be no rows available 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:





## 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

```
UNWIND [0, 1, 2] AS x
CALL {
  WITH x
  RETURN x * 10 AS y
}
RETURN x, y
```

Table 183. Result

x	y
0	0
1	10
2	20

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).

## 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

```
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 184. Result

p.name	p.age
"Alice"	20

p.name	p.age
"Charlie"	65
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 185. Result

p.name	count(other)
"Alice"	2
"Bob"	0
"Charlie"	0
"Dora"	0
Rows: 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 186. Result

p.name	friend
"Alice"	"Bob"
"Bob"	"Alice"

p.name	friend
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:

### Query

```
MATCH (p:Person)
CALL {
  WITH p
  MATCH (p)--(c)
  RETURN count(c) AS numberOfConnections
}
RETURN p.name, numberOfConnections
```

Table 187. Result

p.name	numberOfConnections
"Alice"	2
"Bob"	1
"Charlie"	1
"Dora"	0
Rows: 4	

## 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.

### Query

```
MATCH (p:Person)
CALL {
  WITH p
  UNWIND range (1, 5) AS i
  CREATE (:Person {name: p.name})
}
RETURN count(*)
```

Table 188. Result

count(*)
4

count(\*)

Rows: 1

Nodes created: 20

Properties set: 20

Labels added: 20

## 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

```
MATCH (p:Person)
CALL {
  WITH p
  MATCH (other:Person)
  WHERE other.age < p.age
  RETURN count(other) AS youngerPersonsCount
}
RETURN p.name, youngerPersonsCount
```

Table 189. Result

p.name	youngerPersonsCount
"Alice"	0
"Bob"	1
"Charlie"	3
"Dora"	2

Rows: 4

## 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

```
LOAD CSV FROM 'file:///friends.csv' AS line
CALL {
  WITH line
  CREATE (:PERSON {name: line[1], age: toInteger(line[2])})
} IN TRANSACTIONS
```

Table 190. Result

(empty result)

Rows: 0  
Nodes created: 5  
Properties set: 10  
Labels added: 5  
Transactions committed: 1

As the size of the CSV file in this example is small, only a single separate transaction is started and committed.



`CALL { ... } IN TRANSACTIONS` is only allowed in [implicit transactions](#).

## 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:

friends.csv

```
1,Bill,26
2,Max,27
3,Anna,22
4,Gladys,29
5,Summer,24
```

## 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 191. 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:

#### Query

```
MATCH (n)
CALL {
  WITH n
  DETACH DELETE n
} IN TRANSACTIONS OF 2 ROWS
```

Table 192. Result

(empty result)

Rows: 0  
 Nodes deleted: 9  
 Relationships deleted: 2  
 Transactions committed: 5



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

```
UNWIND [4, 2, 1, 0] AS i
CALL {
  WITH i
  CREATE (:Example {num: 100/i})
} IN TRANSACTIONS OF 2 ROWS
RETURN i
```

## Error

```
/ by zero (Transactions committed: 1)
```

When the failure occurred, the first transaction had already been committed, so the database contains two example nodes.

## Query

```
MATCH (e:Example)
RETURN e.num
```

Table 193. Result

e.num
25
50
Rows: 2

## 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`.

Additionally, there are some restrictions that apply when using an importing `WITH` clause in a `CALL` subquery:

- Only variables imported with the importing `WITH` clause can be used.
- No expressions or aliasing are allowed within the importing `WITH` clause.
- It is not possible to follow an importing `WITH` clause with any of the following clauses: `DISTINCT`, `ORDER BY`, `WHERE`, `SKIP`, and `LIMIT`.

Attempting any of the above, will throw an error. For example, the following query using a `WHERE` clause after an importing `WITH` clause will throw an error:

## Query

```
UNWIND [[1,2],[1,2,3,4],[1,2,3,4,5]] AS l
CALL {
  WITH l
  WHERE size(l) > 2
  RETURN l AS largeLists
}
RETURN largeLists
```

## Error message

Importing WITH should consist only of simple references to outside variables.  
WHERE is not allowed.

A solution to this restriction, necessary for any filtering or ordering of an importing WITH clause, is to declare a second WITH clause after the importing WITH clause. This second WITH clause will act as a regular WITH clause. For example, the following query will not throw an error:

## Query

```
UNWIND [[1,2],[1,2,3,4],[1,2,3,4,5]] AS l
CALL {
  WITH l
  WITH size(l) AS size, l AS l
  WHERE size > 2
  RETURN l AS largeLists
}
RETURN largeLists
```

# CALL procedure

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

## 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 `WHERE` sub-clause followed by a predicate (similar to `WITH ... WHERE ...`).

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

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



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

Users can also develop custom procedures and deploy to the database. See [Java Reference → User-defined procedures](#) for details.

## Call a procedure using `CALL`

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

Query

```
CALL db.labels()
```

Table 194. Result

label
"User"
"Administrator"

Rows: 2

Cypher allows the omission of parentheses on procedures of arity-0 (no arguments).



Best practice is to use parentheses for procedures.

## Query

```
CALL db.labels
```

Table 195. Result

label
"User"
"Administrator"
Rows: 2

## View the signature for a procedure

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

## Query

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

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

Table 196. Result

signature
"dbms.listConfig(searchString = :: STRING?) :: (name :: STRING?, description :: STRING?, value :: STRING?, dynamic :: BOOLEAN?)"
Rows: 1

## Call a procedure using a quoted namespace and name

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

## Query

```
CALL `db`.`labels()`
```

## Query

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

## 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.

## Query

```
CALL dbms.security.createUser('example_username', 'example_password', false)
```

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

## 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 parentheses 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.

## 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.

## 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

```
CALL dbms.security.createUser('example_username', 'example_password')
```

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

## Call a procedure using `CALL YIELD *`

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

### Query

```
CALL db.labels() YIELD *
```

If the procedure has deprecated return columns, those columns are also returned.

## Call a procedure within a complex query using `CALL YIELD`

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

### Query

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

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

## 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 string `'User'`.

Query

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

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

## Call a procedure within a complex query and rename its outputs

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

Query

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

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

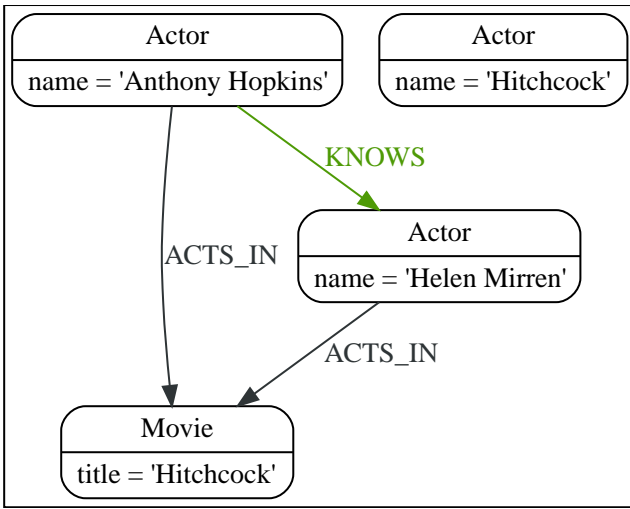
## UNION

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

`UNION` combines the results of two or more queries into a single result set that includes all the rows that belong to any 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.



## Combine two queries and retain duplicates

Combining the results from two queries is done using **UNION ALL**.

Query

```
MATCH (n:Actor)
RETURN n.name AS name
UNION ALL
MATCH (n:Movie)
RETURN n.title AS name
```

The combined result is returned, including duplicates.

Table 197. Result

name
"Anthony Hopkins"
"Helen Mirren"
"Hitchcock"
"Hitchcock"

Rows: 4

## Combine two queries and remove duplicates

By not including **ALL** in the **UNION**, duplicates are removed from the combined result set.

Query

```
MATCH (n:Actor)
RETURN n.name AS name
UNION
MATCH (n:Movie)
RETURN n.title AS name
```

The combined result is returned, without duplicates.

Table 198. Result

name
"Anthony Hopkins"
"Helen Mirren"
"Hitchcock"

Rows: 3

## USE

The **USE** clause determines which graph a query, or query part, is executed against.

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.

## 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 **Fabric**

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 clause**

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](#).

## Examples

### Query a graph by name

In this example we assume that your DBMS contains a database named `myDatabase`:

Query

```
USE myDatabase
MATCH (n) RETURN n
```

### Query a Fabric graph by name [Fabric](#)

In this example we assume that we have configured a Fabric database called `exampleFabricSetup`. The graph that we wish to query is named `exampleDatabaseName`:

Query

```
USE exampleFabricSetup.exampleDatabaseName
MATCH (n) RETURN n
```

### Query a Fabric graph by graph ID [Fabric](#)

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`:

Query

```
USE exampleFabricSetup.graph(0)
MATCH (n) RETURN n
```

## 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`](#)

## 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`.

- <sup>[2]</sup> 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.

## 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.

## 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
```

## 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](https://data.neo4j.com/bands/artists.csv)

```
1,ABBA,1992
2,Roxette,1986
3,Europe,1979
4,The Cardigans,1992
```

## 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
```

## Import data from a CSV file containing headers

When your CSV file has headers, you can view each row in the file as a map instead of as an array of strings.

[artists-with-headers.csv](#)

```
Id,Name,Year
1,ABBA,1992
2,Roxette,1986
3,Europe,1979
4,The Cardigans,1992
```

## Query

```
LOAD CSV WITH HEADERS FROM '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
```

## 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

```
1;ABBA;1992
2;Roxette;1986
3;Europe;1979
4;The Cardigans;1992
```

## 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.

## Result

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

## Importing large amounts of data

If the CSV file contains a significant number of rows (approaching hundreds of thousands or millions), `USING PERIODIC COMMIT` can be used to instruct Neo4j to perform a commit after a number of rows. This reduces the memory overhead of the transaction state. By default, the commit 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](#).



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`.

## Query

```
USING PERIODIC COMMIT LOAD CSV FROM 'file:///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
```

## 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])})
```

## Result

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

## Import data containing escaped characters

In this example, we both have additional quotes around the values, as well as escaped quotes inside one value.

artists-with-escaped-char.csv

```
"1", "The ""Symbol""", "1992"
```

## Query

```
LOAD CSV FROM '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!

## Result

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

## 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.

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

```
+-----+
| number | line |
+-----+
| 1      | ["1","ABBA","1992"] |
| 2      | ["2","Roxette","1986"] |
| 3      | ["3","Europe","1979"] |
| 4      | ["4","The Cardigans","1992"] |
+-----+
4 rows
```

## 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 DISTINCT file() AS path
```

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

```
+-----+
| path                                     |
+-----+
| "/home/example/neo4j/import/artists.csv" |
+-----+
1 row
```

# 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 199. List functions output

Column	Description
<code>name</code>	The name of the function. <span>Default output</span>
<code>category</code>	The function category, for example <code>scalar</code> or <code>string</code> . <span>Default output</span>
<code>description</code>	The function description. <span>Default output</span>
<code>signature</code>	The signature of the function.
<code>isBuiltIn</code>	Whether the function is built-in or user-defined.
<code>argumentDescription</code>	List of the arguments for the function, as map of strings with name, type, default, and description.

Column	Description
<code>returnDescription</code>	The return value type.
<code>aggregating</code>	Whether the function is aggregating or not.
<code>rolesExecution</code>	List of roles permitted to execute this function. Is <code>null</code> without the <code>SHOW ROLE</code> privilege.
<code>rolesBoostedExecution</code>	List of roles permitted to use boosted mode when executing this function. Is <code>null</code> without the <code>SHOW ROLE</code> privilege.

## Syntax



More details about the syntax descriptions can be found [here](#).

List functions, either all or only built-in or user-defined

```
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

```
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

```
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.



When using the `RETURN` clause, the `YIELD` clause is mandatory and must not be omitted.

## 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 *`.



## Query

```
SHOW FUNCTIONS
```

Table 200. Result

name	category	description
"abs"	"Numeric"	"Returns the absolute value of an integer."
"abs"	"Numeric"	"Returns the absolute value of a floating point number."
"acos"	"Trigonometric"	"Returns the arccosine of a number in radians."
"all"	"Predicate"	"Returns true if the predicate holds for all elements in the given list."
"any"	"Predicate"	"Returns true if the predicate holds for at least one element in the given list."
"asin"	"Trigonometric"	"Returns the arcsine of a number in radians."
"atan"	"Trigonometric"	"Returns the arctangent of a number in radians."
"atan2"	"Trigonometric"	"Returns the arctangent2 of a set of coordinates in radians."
"avg"	"Aggregating"	"Returns the average of a set of integer values."
"avg"	"Aggregating"	"Returns the average of a set of floating point values."
"avg"	"Aggregating"	"Returns the average of a set of duration values."
"ceil"	"Numeric"	"Returns the smallest floating point number that is greater than or equal to a number and equal to a mathematical integer."
"coalesce"	"Scalar"	"Returns the first non-null value in a list of expressions."
"collect"	"Aggregating"	"Returns a list containing the values returned by an expression."
"cos"	"Trigonometric"	"Returns the cosine of a number."
"cot"	"Trigonometric"	"Returns the cotangent of a number."
"count"	"Aggregating"	"Returns the number of values or rows."
"date"	"Temporal"	"Create a Date instant."
"date.realtime"	"Temporal"	"Get the current Date instant using the realtime clock."
"date.statement"	"Temporal"	"Get the current Date instant using the statement clock."

Rows: 20

## 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 201. Result

name	isBuiltIn
"abs"	true
"abs"	true
"acos"	true
"all"	true
"any"	true
"asin"	true
"atan"	true
"atan2"	true
"avg"	true
"avg"	true
"avg"	true

Rows: 11

## 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 202. Result

name	category	description	rolesExecution	rolesBoostedExecution	...
"abs"	"Numeric"	"Returns the absolute value of an integer."	<null>	<null>	

name	category	description	rolesExecution	rolesBoostedExecution	...
"abs"	"Numeric"	"Returns the absolute value of a floating point number."	<null>	<null>	
"acos"	"Trigonometric"	"Returns the arccosine of a number in radians."	<null>	<null>	
"all"	"Predicate"	"Returns true if the predicate holds for all elements in the given list."	<null>	<null>	
"any"	"Predicate"	"Returns true if the predicate holds for at least one element in the given list."	<null>	<null>	
"asin"	"Trigonometric"	"Returns the arcsine of a number in radians."	<null>	<null>	
"atan"	"Trigonometric"	"Returns the arctangent of a number in radians."	<null>	<null>	
"atan2"	"Trigonometric"	"Returns the arctangent2 of a set of coordinates in radians."	<null>	<null>	
"avg"	"Aggregating"	"Returns the average of a set of integer values."	<null>	<null>	
"avg"	"Aggregating"	"Returns the average of a set of floating point values."	<null>	<null>	

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 203. Result

name	category	description
"abs"	"Numeric"	"Returns the absolute value of an integer."
"abs"	"Numeric"	"Returns the absolute value of a floating point number."
"acos"	"Trigonometric"	"Returns the arccosine of a number in radians."
"all"	"Predicate"	"Returns true if the predicate holds for all elements in the given list."
"any"	"Predicate"	"Returns true if the predicate holds for at least one element in the given list."
"asin"	"Trigonometric"	"Returns the arcsine of a number in radians."
"atan"	"Trigonometric"	"Returns the arctangent of a number in radians."
"atan2"	"Trigonometric"	"Returns the arctangent2 of a set of coordinates in radians."
"avg"	"Aggregating"	"Returns the average of a set of integer values."
"avg"	"Aggregating"	"Returns the average of a set of floating point values."

Rows: 10

## 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 204. List procedures output

Column	Description
name	The name of the procedure. <span>Default output</span>
description	The procedure description. <span>Default output</span>
mode	The procedure mode, for example <code>READ</code> or <code>WRITE</code> . <span>Default output</span>

Column	Description
<code>worksOnSystem</code>	Whether the procedure can be run on the <code>system</code> database or not. <a href="#">Default output</a>
<code>signature</code>	The signature of the procedure.
<code>argumentDescription</code>	List of the arguments for the procedure, as map of strings with name, type, default, and description.
<code>returnDescription</code>	List of the returned values for the procedure, as map of strings with name, type, and description.
<code>admin</code>	<code>true</code> if this procedure is an admin procedure.
<code>rolesExecution</code>	List of roles permitted to execute this procedure. Is <code>null</code> without the <code>SHOW ROLE</code> privilege.
<code>rolesBoostedExecution</code>	List of roles permitted to use boosted mode when executing this procedure. Is <code>null</code> without the <code>SHOW ROLE</code> privilege.
<code>option</code>	Map of extra output, e.g. if the procedure is deprecated.

## Syntax



More details about the syntax descriptions can be found [here](#).

### 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.

## 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 205. Result

name	description	mode	worksOnSystem
"db.awaitIndex"	"Wait for an index to come online (for example: CALL db.awaitIndex("MyIndex", 300))."	"READ"	true
"db.awaitIndexes"	"Wait for all indexes to come online (for example: CALL db.awaitIndexes(300))."	"READ"	true
"db.checkpoint"	"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."	"DBMS"	true
"db.clearQueryCaches"	"Clears all query caches."	"DBMS"	true
"db.constraints"	"List all constraints in the database."	"READ"	true
"db.createIndex"	"Create a named schema index with specified index provider and configuration (optional). Yield: name, labels, properties, providerName, status"	"SCHEMA"	false
"db.createLabel"	"Create a label"	"WRITE"	false

name	description	mode	worksOnSystem
"db.createNodeKey"	"Create a named node key constraint. Backing index will use specified index provider and configuration (optional). Yield: name, labels, properties, providerName, status"	"SCHEMA"	false
"db.createProperty"	"Create a Property"	"WRITE"	false
"db.createRelationshipType"	"Create a RelationshipType"	"WRITE"	false
"db.createUniquePropertyConstraint"	"Create a named unique property constraint. Backing index will use specified index provider and configuration (optional). Yield: name, labels, properties, providerName, status"	"SCHEMA"	false
"db.index.fulltext.awaitEventuallyConsistentIndexRefresh"	"Wait for the updates from recently committed transactions to be applied to any eventually-consistent full-text indexes."	"READ"	true
"db.index.fulltext.createNodeIndex"	"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."	"SCHEMA"	false

name	description	mode	worksOnSystem
"db.index.fulltext.createRelationshipIndex"	"Create a relationship full-text index for the given relationship types 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."	"SCHEMA"	false
"db.index.fulltext.drop"	"Drop the specified index."	"SCHEMA"	false

Rows: 15

## 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

```
SHOW PROCEDURES YIELD name, admin
WHERE admin
```

Table 206. Result

name	admin
"db.clearQueryCaches"	true
"db.listLocks"	true
"db.prepareForReplanning"	true
"db.stats.clear"	true
"db.stats.collect"	true
"db.stats.retrieve"	true
"db.stats.retrieveAllAnonymized"	true

Rows: 7



## 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 207. Result

name	description	rolesExecution	rolesBoostedExecution	...
"db.awaitIndex"	"Wait for an index to come online (for example: CALL db.awaitIndex("MyIndex", 300))."	<null>	<null>	
"db.awaitIndexes"	"Wait for all indexes to come online (for example: CALL db.awaitIndexes(300))."	<null>	<null>	
"db.checkpoint"	"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."	<null>	<null>	
"db.constraints"	"List all constraints in the database."	<null>	<null>	
"db.createIndex"	"Create a named schema index with specified index provider and configuration (optional). Yield: name, labels, properties, providerName, status"	<null>	<null>	
"db.createLabel"	"Create a label"	<null>	<null>	

name	description	rolesExecution	rolesBoostedExecution	...
"db.createNodeKey"	"Create a named node key constraint. Backing index will use specified index provider and configuration (optional). Yield: name, labels, properties, providerName, status"	<null>	<null>	
"db.createProperty"	"Create a Property"	<null>	<null>	
"db.createRelationshipType"	"Create a RelationshipType"	<null>	<null>	
"db.createUniquePropertyConstraint"	"Create a named unique property constraint. Backing index will use specified index provider and configuration (optional). Yield: name, labels, properties, providerName, status"	<null>	<null>	

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 208. Result

name	description	mode	worksOnSystem
"db.awaitIndex"	"Wait for an index to come online (for example: CALL db.awaitIndex("MyIndex", 300))."	"READ"	true
"db.awaitIndexes"	"Wait for all indexes to come online (for example: CALL db.awaitIndexes(300))."	"READ"	true

name	description	mode	worksOnSystem
"db.checkpoint"	"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."	"DBMS"	true
"db.constraints"	"List all constraints in the database."	"READ"	true
"db.createIndex"	"Create a named schema index with specified index provider and configuration (optional). Yield: name, labels, properties, providerName, status"	"SCHEMA"	false
"db.createLabel"	"Create a label"	"WRITE"	false
"db.createNodeKey"	"Create a named node key constraint. Backing index will use specified index provider and configuration (optional). Yield: name, labels, properties, providerName, status"	"SCHEMA"	false
"db.createProperty"	"Create a Property"	"WRITE"	false
"db.createRelationshipType"	"Create a RelationshipType"	"WRITE"	false
"db.createUniquePropertyConstraint"	"Create a named unique property constraint. Backing index will use specified index provider and configuration (optional). Yield: name, labels, properties, providerName, status"	"SCHEMA"	false

Rows: 10

[2] See <https://developer.mozilla.org/en-US/docs/Glossary/percent-encoding>

# Values and types

Cypher supports a range of data values. When writing Cypher queries, it is not possible to declare a data type. Rather, Cypher will automatically infer the data type of a given value.

More information about the data values and types supported by Cypher can be found in the following sections:

- [Property, structural, and composite values](#)
- [Temporal values](#)
- [Spatial values](#)
- [Working with null](#)
- [Lists](#)
- [Maps](#)
- [Casting data values](#)

## Property, structural, and composite values

Cypher provides first class support for a number of data value types. These fall into the following three categories: **property**, **structural**, and **composite**. This section will first provide a brief overview of each type, and then go into more detail about the property data type.

### Property types

A property type value is one that can be stored as a node or relationship property.

The following data types are included in the property types category: **Boolean**, **Date**, **DateTime**, **Duration**, **Float**, **Integer**, **List**, **LocalTime**, **LocalDateTime**, **Point**, **String**, and **Time**.

- Property types can be returned from Cypher queries.
- Property types can be used as [parameters](#).
- Property types can be stored as properties.
- Property types can be constructed with [Cypher literals](#).

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

Cypher also provides pass-through support for byte arrays, which can be stored as property values. Byte arrays are supported for performance reasons, since using Cypher's generic language type, List of Integer (where each Integer has a 64-bit representation), would be too costly. However, byte arrays are not considered a first class data type by Cypher, so they do not have a literal representation.

## Structural types

The following data types are included in the structural types category: **Node**, **Relationship**, and **Path**.

- Structural types can be returned from Cypher queries.
- Structural types cannot be used as [parameters](#).
- Structural types cannot be stored as properties.
- Structural types cannot be constructed with [Cypher literals](#).

The **Node** data type includes: Id, Label(s), and a Map (of properties). Note that labels are not values, but a form of pattern syntax.

The **Relationship** data type includes: Id, Type, Map (of properties), start node Id, and end node Id.

The **Path** data type is an alternating sequence of nodes and relationships.



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.

## Composite types

The following data types are included in the composite types category: **List** and **Map**.

- Composite types can be returned from Cypher queries.
- Composite types can be used as [parameters](#).
- Composite types cannot be stored as properties (with the exception of homogenous lists).
- Composite types can be constructed with [Cypher literals](#).

The **List** data type can be either a homogenous collection of simple values, or a heterogeneous, ordered collection of values, each of which can have any property, structural or composite type.

The **Map** data type is a heterogeneous, unordered collection of (Key, Value) pairs, where Key is a literal and Value can have any property, structural, or composite type.

Composite values can also contain **null**. For more details, see [working with null](#).

## Property type details

The below table provides more detailed information about the various property types that Cypher supports. Note that Cypher types are implemented using Java, and that below table references Java value constants.

Type	Min. value	Max. value	Precision
<b>Boolean</b>	<b>False</b>	<b>True</b>	-

Type	Min. value	Max. value	Precision
Date	-999_999_999-01-01	+999_999_999-12-31	Days
DateTime	-999_999_999-01-01T00:00:00+18:00	+999_999_999-12-31T23:59:59.999999999-18:00	Nanoseconds
Duration	P-292471208677Y-6M-15DT-15H-36M-32S	P292471208677Y6M15DT15H36M32.999999999S	Nanoseconds
Float	Double.MIN_VALUE <sup>[3]</sup>	Double.MAX_VALUE	64 bit
Integer	Long.MIN_VALUE	Long.MAX_VALUE	64 bit
LocalDateTime	-999_999_999-01-01T00:00:00	+999_999_999-12-31T23:59:59.999999999	Nanoseconds
LocalTime	00:00:00	23:59:59.999999999	Nanoseconds
Point	<p><b>Cartesian:</b> (-Double.MAX_VALUE, -Double.MAX_VALUE)</p> <p><b>Cartesian_3D:</b> (-Double.MAX_VALUE, -Double.MAX_VALUE, -Double.MAX_VALUE)</p> <p><b>WGS_84:</b> (-180, -90)</p> <p><b>WGS_84_3D:</b> (-180, -90, -Double.MAX_VALUE)</p>	<p><b>Cartesian:</b> (Double.MAX_VALUE, Double.MAX_VALUE)</p> <p><b>Cartesian_3D:</b> (Double.MAX_VALUE, Double.MAX_VALUE, Double.MAX_VALUE)</p> <p><b>WGS_84:</b> (180, 90)</p> <p><b>WGS_84_3D:</b> (180, 90, Double.MAX_VALUE)</p>	The precision of each coordinate of the Point is 64 bit as they are floats.
String	-	-	-
Time	00:00:00+18:00	23:59:59.999999999-18:00	Nanoseconds

## Java value details

Name	Value
Double.MAX_VALUE	1.7976931348623157e+308
Double.MIN_VALUE	4.9e-324
Long.MAX_VALUE	2 <sup>63</sup> -1
Long.MIN_VALUE	-2 <sup>63</sup>

## Temporal values

Cypher has built-in support for handling temporal values, which can be stored as properties on nodes and relationships in Neo4j databases. This section will discuss how Cypher handles time zones, before exploring temporal values in more detail.



- 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.

## Temporal value types

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

Type	Date support	Time support	Time zone support
<code>Date</code>	✓		
<code>Time</code>		✓	✓
<code>LocalTime</code>		✓	
<code>DateTime</code>	✓	✓	✓
<code>LocalDateTime</code>	✓	✓	
<code>Duration</code>	-	-	-

`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.

## 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.

## 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.

Temporal instant type	Composition of parts
<code>Date</code>	<code>&lt;date&gt;</code>
<code>Time</code>	<code>&lt;time&gt;&lt;timezone&gt;</code> or <code>T&lt;time&gt;&lt;timezone&gt;</code>
<code>LocalTime</code>	<code>&lt;time&gt;</code> or <code>T&lt;time&gt;</code>
<code>DateTime*</code>	<code>&lt;date&gt;T&lt;time&gt;&lt;timezone&gt;</code>
<code>LocalDateTime*</code>	<code>&lt;date&gt;T&lt;time&gt;</code>

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

### Specifying dates

Component	Format	Description
Year	<code>YYYY</code>	Specified with at least four digits ( <a href="#">special rules apply in certain cases</a> ).
Month	<code>MM</code>	Specified with a double digit number from <code>01</code> to <code>12</code> .



Component	Format	Description
Week	ww	Always prefixed with <b>W</b> and specified with a double digit number from <b>01</b> to <b>53</b> .
Quarter	q	Always prefixed with <b>Q</b> and specified with a single digit number from <b>1</b> to <b>4</b> .
Day of the month	DD	Specified with a double digit number from <b>01</b> to <b>31</b> .
Day of the week	D	Specified with a single digit number from <b>1</b> to <b>7</b> .
Day of the quarter	DD	Specified with a double digit number from <b>01</b> to <b>92</b> .
Ordinal day of the year	DDD	Specified with a triple digit number from <b>001</b> to <b>366</b> .

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:

Format	Description	Example	Interpretation of example
YYYY-MM-DD	Calendar date: Year-Month-Day	2015-07-21	2015-07-21
YYYYMMDD	Calendar date: Year-Month-Day	20150721	2015-07-21
YYYY-MM	Calendar date: Year-Month	2015-07	2015-07-01
YYYYMM	Calendar date: Year-Month	201507	2015-07-01
YYYY-Www-D	Week date: Year-Week-Day	2015-W30-2	2015-07-21
YYYYWwwD	Week date: Year-Week-Day	2015W302	2015-07-21
YYYY-Www	Week date: Year-Week	2015-W30	2015-07-20
YYYYWww	Week date: Year-Week	2015W30	2015-07-20

Format	Description	Example	Interpretation of example
YYYY-Qq-DD	Quarter date: Year-Quarter-Day	2015-Q2-60	2015-05-30
YYYYQqDD	Quarter date: Year-Quarter-Day	2015Q260	2015-05-30
YYYY-Qq	Quarter date: Year-Quarter	2015-Q2	2015-04-01
YYYYQq	Quarter date: Year-Quarter	2015Q2	2015-04-01
YYYY-DDD	Ordinal date: Year-Day	2015-202	2015-07-21
YYYYDDD	Ordinal date: Year-Day	2015202	2015-07-21
YYYY	Year	2015	2015-01-01

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

## Specifying times

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

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:

Format	Description	Example	Interpretation of example
HH:MM:SS.ssssssss	Hour:Minute:Second.fraction	21:40:32.142	21:40:32.142
HHMMSS.ssssssss	Hour:Minute:Second.fraction	214032.142	21:40:32.142
HH:MM:SS	Hour:Minute:Second	21:40:32	21:40:32.000
HHMMSS	Hour:Minute:Second	214032	21:40:32.000

Format	Description	Example	Interpretation of example
HH:MM	Hour:Minute	21:40	21:40:00.000
HHMM	Hour:Minute	2140	21:40:00.000
HH	Hour	21	21:00:00.000

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

## Specifying time zones

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

- As an offset from UTC.
- Using the **Z** shorthand for the UTC ( $\pm 00:00$ ) time zone.

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

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

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

The following formats are supported for specifying time zones:

Format	Description	Example	Supported for <i>DateTime</i>	Supported for <i>Time</i>
Z	UTC	Z	✓	✓
$\pm$ HH:MM	Hour:Minute	+09:30	✓	✓
$\pm$ HH:MM[ZoneName]	Hour:Minute[ZoneName]	+08:45[Australia/EucLa]	✓	
$\pm$ HHMM	Hour:Minute	+0100	✓	✓
$\pm$ HHMM[ZoneName]	Hour:Minute[ZoneName]	+0200[Africa/Johannesburg]	✓	
$\pm$ HH	Hour	-08	✓	✓
$\pm$ HH[ZoneName]	Hour[ZoneName]	+08[Asia/Singapore]	✓	

Format	Description	Example	Supported for DateTime	Supported for Time
[ZoneName]	[ZoneName]	[America/Regina]	✓	

## Components of temporal instants

Components of temporal instant values can be accessed as properties.

Table 209. Components of temporal instant values and where they are supported

Component	Description	Type	Range/Format	Date	DateTime	LocalDateTime	Time	LocalTime
<code>instant.year</code>	The <code>year</code> component represents the <a href="#">astronomical year number</a> of the instant. <sup>[4]</sup>	Integer	At least 4 digits. For more information, see the <a href="#">rules for using the Year component</a> .	✓	✓	✓		
<code>instant.quarter</code>	The <code>quarter-of-the-year</code> component.	Integer	1 to 4.	✓	✓	✓		
<code>instant.month</code>	The <code>month-of-the-year</code> component.	Integer	1 to 12.	✓	✓	✓		
<code>instant.week</code>	The <code>week-of-the-year</code> component. <sup>[5]</sup>	Integer	1 to 53.	✓	✓	✓		
<code>instant.weekYear</code>	The year that the <code>week-of-year</code> component belongs to. <sup>[6]</sup>	Integer	At least 4 digits. For more information, see the <a href="#">rules for using the Year component</a> .	✓	✓	✓		
<code>instant.dayOfQuarter</code>	The <code>day-of-the-quarter</code> component.	Integer	1 to 92.	✓	✓	✓		
<code>instant.quarterDay</code>	The <code>day-of-the-quarter</code> component (alias for <code>instant.dayOfQuarter</code> ).	Integer	1 to 92.	✓	✓	✓		
<code>instant.day</code>	The <code>day-of-the-month</code> component.	Integer	1 to 31.	✓	✓	✓		
<code>instant.ordinalDay</code>	The <code>day-of-the-year</code> component.	Integer	1 to 366.	✓	✓	✓		

Component	Description	Type	Range/Format	Date	DateTime	LocalDateTime	Time	LocalTime
<code>instant.dayOfWeek</code>	The day-of-the-week component (the first day of the week is Monday).	Integer	1 to 7.	✓	✓	✓		
<code>instant.weekDay</code>	The day-of-the-week component (alias for <code>instant.dayOfWeek</code> ).	Integer	1 to 7.	✓	✓	✓		
<code>instant.hour</code>	The hour component.	Integer	0 to 23.		✓	✓	✓	✓
<code>instant.minute</code>	The minute component.	Integer	0 to 59.		✓	✓	✓	✓
<code>instant.second</code>	The second component. <sup>[7]</sup>	Integer	0 to 59.		✓	✓	✓	✓
<code>instant.millisecond</code>	The millisecond component.	Integer	0 to 999.		✓	✓	✓	✓
<code>instant.microsecond</code>	The microsecond component.	Integer	0 to 999999.		✓	✓	✓	✓
<code>instant.nanosecond</code>	The nanosecond component.	Integer	0 to 999999999.		✓	✓	✓	✓
<code>instant.timezone</code>	The timezone component.	String	Depending on how the <a href="#">time zone was specified</a> , this is either a time zone name or an offset from UTC in the format <code>±HHMM</code> .		✓		✓	
<code>instant.offset</code>	The timezone offset.	String	In the format <code>±HHMM</code> .		✓		✓	
<code>instant.offsetMinutes</code>	The timezone offset in minutes.	Integer	-1080 to +1080.		✓		✓	
<code>instant.offsetSeconds</code>	The timezone offset in seconds.	Integer	-64800 to +64800.		✓		✓	
<code>instant.epochMillis</code>	The number of milliseconds between <code>1970-01-01T00:00:00+0000</code> and the instant. <sup>[8]</sup>	Integer	Positive for instants after and negative for instants before <code>1970-01-01T00:00:00+0000</code> .		✓			

Component	Description	Type	Range/Format	Date	DateTime	LocalDateTime	Time	LocalTime
<code>instant.epochSeconds</code>	The number of seconds between <code>1970-01-01T00:00:00+0000</code> and the instant. <sup>[9]</sup>	Integer	Positive for instants after and negative for instants before <code>1970-01-01T00:00:00+0000</code> .		✓			

## Examples

Below are examples of parsing instant values using various temporal functions. More information about these temporal functions can be found [here](#).

### Example 1. datetime

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

Query

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

Table 210. Result

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

### Example 2. localdatetime

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

Query

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

Table 211. Result

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

### Example 3. date

Parsing a Date using the week date format:

Query

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

Table 212. Result

theDate
2015-03-26
Rows: 1

### Example 4. time

Parsing a Time:

Query

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

Table 213. Result

theTime
12:50:35.556000000+01:00
Rows: 1

### Example 5. localtime

Parsing a LocalTime:

Query

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

Table 214. Result

theLocalTime
12:50:35.556000000
Rows: 1

## Example 6. date

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 215. Result

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

## Example 7. datetime

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

### Query

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

Table 216. Result

d.year	d.quarter	d.month	d.week	d.weekYear	d.day	d.ordinalDay	d.dayOfWeek	d.dayOfQuarter
1984	4	11	45	1984	11	316	7	42
Rows: 1								



### Example 8. datetime

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

Query

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

Table 217. Result

d.hour	d.minute	d.second	d.millisecond	d.microsecond	d.nanosecond
12	31	14	645	645876	645876123
Rows: 1					

### Example 9. datetime

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

Query

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

Table 218. Result

d.timezone	d.offset	d.offsetMinutes	d.epochSeconds	d.epochMillis
"Europe/Stockholm"	"+01:00"	60	469020674	469020674645
Rows: 1				

## Example 10. date.truncate

Get the first Date of the current year:

Query

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

Table 219. Result

day
2022-01-01
Rows: 1

## Example 11. date.truncate

Get the Date of the Thursday in the week of a specific date:

Query

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

Table 220. Result

thursday
2019-10-03
Rows: 1

## 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:

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

## 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:

Component Group	Component	Description	Type	Details
Months	<b>duration.years</b>	The total number of years.	Integer	Each set of <b>4</b> quarters is counted as <b>1</b> year; each set of <b>12</b> months is counted as <b>1</b> year.
	<b>duration.quarters</b>	The total number of quarters.	Integer	Each year is counted as <b>4</b> quarters; each set of <b>3</b> months is counted as <b>1</b> quarter.
	<b>duration.months</b>	The total number of months.	Integer	Each year is counted as <b>12</b> months; each <code>quarter_</code> is counted as <b>3</b> months.
Days	<b>duration.weeks</b>	The total number of weeks.	Integer	Each set of <b>7</b> days is counted as <b>1</b> week.
	<b>duration.days</b>	The total number of days.	Integer	Each week is counted as <b>7</b> days.

Component Group	Component	Description	Type	Details
Seconds	<code>duration.hours</code>	The total number of hours.	Integer	Each set of 60 minutes is counted as 1 hour; each set of 3600 seconds is counted as 1 hour.
	<code>duration.minutes</code>	The total number of minutes.	Integer	Each hour is counted as 60 minutes; each set of 60 seconds is counted as 1 minute.
	<code>duration.seconds</code>	The total number of seconds.	Integer	Each hour is counted as 3600 seconds; each minute is counted as 60 seconds.
	<code>duration.milliseconds</code>	The total number of milliseconds	Integer	Each set of 1000 milliseconds is counted as 1 second.
	<code>duration.microseconds</code>	The total number of microseconds.	Integer	Each millisecond is counted as 1000 microseconds.
	<code>duration.nanoseconds</code>	The total number of nanoseconds.	Integer	Each microsecond is counted as 1000 nanoseconds.

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:

Component	Component Group	Description	Type
<code>duration.quartersOfYear</code>	Months	The number of quarters in the group that do not make a whole year.	Integer
<code>duration.monthsOfYear</code>	Months	The number of months in the group that do not make a whole year.	Integer
<code>duration.monthsOfQuarter</code>	Months	The number of months in the group that do not make a whole quarter.	Integer
<code>duration.daysOfWeek</code>	Days	The number of days in the group that do not make a whole week.	Integer
<code>duration.minutesOfHour</code>	Seconds	The number of minutes in the group that do not make a whole hour.	Integer
<code>duration.secondsOfMinute</code>	Seconds	The number of seconds in the group that do not make a whole minute.	Integer

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

## Examples

Below are examples of parsing durations using the `duration()` function. More information can be found [here](#).

### Example 12. duration

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

Query

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

Table 221. Result

theDuration
P14DT16H12M
Rows: 1

### Example 13. duration

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

Query

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

Table 222. Result

theDuration
P5M1DT12H
Rows: 1

### Example 14. duration

Return a Duration of 45 seconds:

Query

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

Table 223. Result

theDuration
PT45S
Rows: 1

### Example 15. duration

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

Query

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

Table 224. Result

theDuration
P17DT12H
Rows: 1

### Example 16. duration

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

Query

```
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 225. Result

d.years	d.quarters	d.quartersOfYear	d.months	d.monthsOfYear	d.monthsOfQuarter
1	5	1	17	5	2
Rows: 1					

### Example 17. duration

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

Query

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

Table 226. Result

d.weeks	d.days	d.daysOfWeek
3	25	4
Rows: 1		

### Example 18. duration

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

Query

```
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 227. Result

d.hours	d.minutes	d.seconds	d.milliseconds	d.microseconds	d.nanoseconds
1	61	3661	3661111	3661111111	3661111111111
Rows: 1					

### Example 19. duration

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

Query

```
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 228. Result

d.minutesOfHour	d.secondsOfMinute	d.millisecondsOfSecond	d.microsecondsOfSecond	d.nanosecondsOfSecond
1	1	111	111111	111111111

Rows: 1

### Example 20. duration

Create a Duration representing 1.5 days:

Query

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

Table 229. Result

theDuration
P1DT12H

Rows: 1

### Example 21. duration.between

Compute the Duration between two temporal instants:

Query

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

Table 230. Result

theDuration
P30Y8M13D

Rows: 1



### Example 22. duration.inDays

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

Query

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

Table 231. Result

theDuration
P299D
Rows: 1

### Example 23. date.truncate

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

Query

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

Table 232. Result

lastDay
2022-07-31
Rows: 1

### Example 24. time

Add a *Duration* to a *Date*:

Query

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

Table 233. Result

theTime
01:42:19Z
Rows: 1

### Example 25. duration

Add two Duration values:

Query

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

Table 234. Result

theDuration
P1M2DT25H
Rows: 1

### Example 26. duration

Multiply a Duration by a number:

Query

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

Table 235. Result

theDuration
PT74H54M
Rows: 1

### Example 27. duration

Divide a Duration by a number:

Query

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

Table 236. Result

theDuration
PT1H38M
Rows: 1

## Example 28. datetime

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 237. Result

lessThanOneDayApart
true
Rows: 1

## Example 29. date

Return the abbreviated name of the current month:

Query

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

Table 238. Result

month
"Jun"
Rows: 1

## 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.

## Spatial values

Cypher has built-in support for handling spatial values (points), which can be stored as properties on nodes and relationships in Neo4j databases.

This section begins with an explanation of the `Point` type. It then proceeds to discuss Cypher's support of Coordinate Reference Systems, and how to work with spatial instants in Cypher, including how spatial point instants work with Cypher indexing. Finally, it briefly explains comparability and orderability with

regard to spatial instants.



For more information about spatial functions, allowing for the creation and manipulation of spatial values, see the section on [Spatial functions](#).

For more information about the comparison and ordering of spatial values, see the section on the [ordering and comparison of values](#).

## The Point type

Neo4j supports the `Point` type for values of spatial geometry.

Values with the `Point` type have 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 and relationships with `Point` or `List(Point)` properties can be indexed using a [point index](#). This is true for all CRSs (and for both 2D and 3D).
- 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).

## 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:

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 but of a different dimension. For example, if you search for 3D points using a 2D range, you will get no results. However, they can be ordered, as discussed in more detail in the section about [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\(WGS-84-3D\)](#).

## Converting coordinate units

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, such as '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 those supported, it is necessary to explicitly convert them.

For example, if the incoming `$height` is a string field in kilometers, it would be necessary to add `height: toFloat($height) * 1000` to the query. Likewise if the results of the `distance` function are expected to be returned in kilometers, an explicit conversion is required. The below query is an example of this conversion:

Query

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

Table 239. Result

km
42
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. This means that when two Cartesian points are passed to the `distance` function, the resulting value will be in the same units as the original coordinates. This is true for both 2D and 3D points, as the Pythagoras equation used is generalized to any number of dimensions. However, just as you cannot compare geographic points to Cartesian points, you cannot calculate the distance between a 2D point and a 3D point. If you need to do that, explicitly transform the one type into the other. For example:

Query

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

Table 240. Result

bad	good
<null>	5.0
Rows: 1	

## Spatial instants

All point types are created from two components:

- The `Coordinate` containing either 2 or 3 floating point values (64-bit).
- The `Coordinate Reference System` (or `CRS`) defining the meaning (and possibly units) of the values in the `Coordinate`.

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

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

All fields are provided to the `point` function in the form of a map of explicitly named arguments. Neo4j does not support an ordered list of coordinate fields because of the contradictory conventions between geographic and cartesian coordinates, where geographic coordinates normally list `y` before `x` (`latitude` before `longitude`).

The following query which returns points created in each of the four supported `CRS`s. Take particular note

of the order and keys of the coordinates in the original `point` function, and how those values are displayed in the results:

### Query

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

Table 241. Result

cartesian_2d	cartesian_3d	geo_2d	geo_3d
<code>point({srid:7203, x: 3.0, y: 0.0})</code>	<code>point({srid:9157, x: 0.0, y: 4.0, z: 1.0})</code>	<code>point({srid:4326, x: 56.0, y: 12.0})</code>	<code>point({rid:4979, x: 56.0, y: 12.0, z: 1000.0})</code>
Rows: 1			

For the geographic coordinates, it is important to note that the `latitude` value should always lie in the interval `[-90, 90]`. Any other value outside this range will throw an exception. The `longitude` value should always lie in the interval `[-180, 180]`. 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. Any value within the allowed range of the signed 64-bit floating point type will be accepted.

## Components of points

Components of point values can be accessed as properties.

Table 242. Components of point instances and where they are supported

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

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

## Examples

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

### Query

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

Table 243. Result

x	y	crs	srid
<code>3.0</code>	<code>4.0</code>	<code>"cartesian"</code>	<code>7203</code>



x	y	crs	srid
Rows: 1			

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

Query

```
WITH point({latitude: 3, longitude: 4, height: 4321}) AS p
RETURN
  p.latitude 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 244. Result

latitude	longitude	height	x	y	z	crs	srid
3.0	4.0	4321.0	4.0	3.0	4321.0	"wgs-84-3d"	4979
Rows: 1							

## Spatial values and indexes

If there is a RANGE or POINT index on a particular node or relationship property, and a spatial point is assigned to that property on a node or relationship, the node or relationship will be indexed.

In a POINT index, Neo4j uses space filling curves in 2D or 3D over an underlying generalized B+Tree. This index has support for [equality](#), [distance](#), and [bounding box](#) queries.

In a RANGE index, the points will be sorted according to their lexicographic ordering per coordinate reference system. For point values, this index has support for [equality](#).

## Comparability and orderability

Cypher does not support comparing spatial values using the inequality operators, `<`, `<=`, `>`, and `>=`. Attempting to do so will return `null`.

To compare spatial points within a specific range, instead use the spatial functions [point.distance](#) or [point.withinBBox](#).

## Working with `null`

In Cypher, `null` is used to represent missing or undefined values. All data types in Cypher are nullable.

Conceptually, `null` means a missing or unknown value, and it is treated somewhat differently from other values. For example, returning a property from a node that does not have said property produces `null`. Most expressions that take `null` as input will produce `null`. In the case of a predicate used in a `WHERE` clause, 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. This means that the expression `null = null` yields `null`, and not `true`.

## Logical operations with `null`

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

Table 245. Truth table for logical operators

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

## The `IN` operator and `null`

The **IN operator** follows similar logic. If Cypher can ascertain that something exists in a list, the result will be `true`. Any list that contains a `null` and does not have a matching element will return `null`. Otherwise, the result will be `false`.

Table 246. examples of expressions containing the `IN` operator

Expression	Result
<code>2 IN [1, 2, 3]</code>	<code>true</code>
<code>2 IN [1, null, 3]</code>	<code>null</code>
<code>2 IN [1, 2, null]</code>	<code>true</code>
<code>2 IN [1]</code>	<code>false</code>
<code>2 IN []</code>	<code>false</code>
<code>null IN [1, 2, 3]</code>	<code>null</code>
<code>null IN [1, null, 3]</code>	<code>null</code>
<code>null IN []</code>	<code>false</code>

Using `all`, `any`, `none`, and `single` follows a similar rule. If the result can be calculated definitively, `true` or `false` is returned. Otherwise `null` is produced.

## The `[]` operator and `null`

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

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

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

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

## Expressions that return `null`

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

## Using `IS NULL` and `IS NOT NULL`

Testing any value against `null`, either with the `=` operator or with the `<>` operator, always evaluates to `null`. Therefore, use the special equality operators `IS NULL` or `IS NOT NULL`.

## Lists

Cypher includes comprehensive support for lists. This section first describes lists in general, and then discusses how to use list comprehension and pattern comprehension in lists.



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](#).

## 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 247. Result

list
[0,1,2,3,4,5,6,7,8,9]
Rows: 1

A list can consist of different value types.

Query

```
RETURN [0, "hello", 3.14, null] AS list
```

Table 248. Result

list
[0, "hello", 3.14, null]
Rows: 1

Lists are indexed by 0 in Cypher. To access individual elements in a list, use square brackets. This extracts from the start index and up to, but not including, the end index.

For example:

Query

```
---
WITH [5,1,7] AS list
RETURN list[2]
---
```

Table 249. Result

list[2]
7
Rows: 1

## List range and size

The below examples use the `range` function to create lists. This function returns a list containing all numbers between given start and end numbers. The range is inclusive in both ends.

Query

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

Table 250. Result

list
3
Rows: 1

It is also possible to use negative numbers, to start from the end of the list instead.

Query

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

Table 251. Result

list
8
Rows: 1

Finally, it is possible to use ranges inside the brackets to return ranges of the list. The list range operator (`[]`) is inclusive of the first value, but exclusive of the last value.

Query

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

Table 252. Result

list
[0,1,2]
Rows: 1

Query

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

Table 253. Result

list
[0,1,2,3,4,5]
Rows: 1

Query

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

Table 254. Result

list
[6,7,8,9,10]
Rows: 1

Query

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

Table 255. Result

list
[0,1,2,3]
Rows: 1

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

Query

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

Table 256. Result

list
<null>
Rows: 1

Query

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

Table 257. Result

list
[5,6,7,8,9,10]
Rows: 1

The `size` of a list can be obtained as follows:

Query

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

Table 258. Result

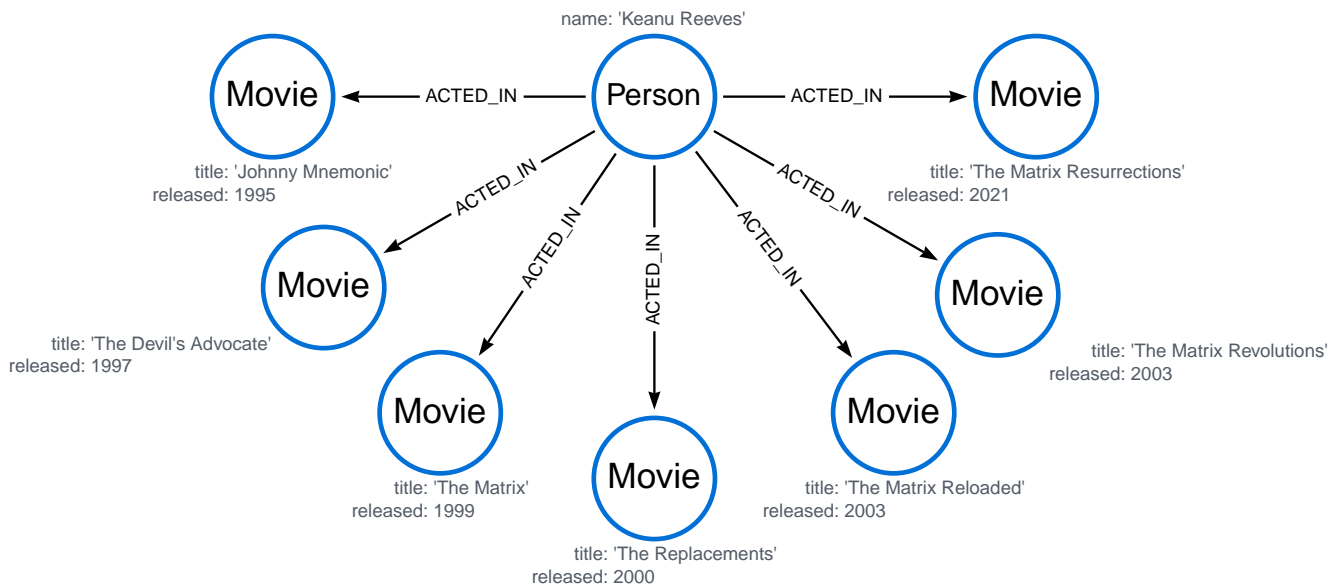
list
3
Rows: 1

## 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.

## Example graph

The following graph is used for examples below:



To recreate the graph, run the following query against an empty Neo4j database:

```
CREATE
(keanu:Person {name: 'Keanu Reeves'}),
(johnnyMnemonic:Movie {title: 'Johnny Mnemonic', released: 1995}),
(theMatrixRevolutions:Movie {title: 'The Matrix Revolutions', released: 2003}),
(theMatrixReloaded:Movie {title: 'The Matrix Reloaded', released: 2003}),
(theReplacements:Movie {title: 'The Replacements', released: 2000}),
(theMatrix:Movie {title: 'The Matrix', released: 1999}),
(theDevilsAdvocate:Movie {title: 'The Devils Advocate', released: 1997}),
(theMatrixResurrections:Movie {title: 'The Matrix Resurrections', released: 2021}),
(keanu)-[:ACTED_IN]->(johnnyMnemonic),
(keanu)-[:ACTED_IN]->(theMatrixRevolutions),
(keanu)-[:ACTED_IN]->(theMatrixReloaded),
(keanu)-[:ACTED_IN]->(theReplacements),
(keanu)-[:ACTED_IN]->(theMatrix),
(keanu)-[:ACTED_IN]->(theDevilsAdvocate),
(keanu)-[:ACTED_IN]->(theMatrixResurrections)
```

## Examples

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

### Query

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

Table 259. Result

years
[2021, 2003, 2003, 1999]
Rows: 1

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

## Storing lists as properties

It is possible to store homogenous lists of simple values as properties. For example, the following query creates a list from the **title** properties of the **Movie** nodes connected to **Keanu Reeves**. It then sets that list as a **resume** property on **Keanu Reeves**.

### Query

```
MATCH (keanu:Person {name: 'Keanu Reeves'})
WITH keanu, [(keanu)-->(b:Movie) | b.title] AS movieTitles
SET keanu.resume = movieTitles
RETURN keanu.resume
```

Table 260. Result

keanu.resume
["The Matrix Resurrections", "The Devils Advocate", "The Matrix", "The Replacements", "The Matrix Reloaded", "The Matrix Revolutions", "Johnny Mnemonic"]
Rows: 1

It is not, however, possible to store heterogenous lists as properties. For example, the following query, which tries to set a list including both the **title** and the **released** properties as the **resume** property of **Keanu Reeves** will fail. This is because the **title** property values are stored as strings, while the **released** property values are stored as integers.

### Query

```
MATCH (keanu:Person {name: 'Keanu Reeves'})
WITH keanu, [(keanu)-->(b:Movie) | b.title] + [(keanu)-->(b:Movie) | b.released] AS movieTitles
SET keanu.resume = movieTitles
RETURN keanu.resume
```

Neo4j only supports a subset of Cypher types for storage as singleton or array properties. Please refer to section `cypher/syntax/values` of the manual for more details.

## List comprehension

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

For example, the following query returns a new list from the previously created **resume** property (a list of strings) of **Keanu Reeves**:

### Query

```
MATCH (keanu:Person {name: 'Keanu Reeves'})
RETURN [x IN keanu.resume WHERE x contains 'The Matrix'] AS matrixList
```

Table 261. Result



matrixList

```
["The Matrix Resurrections", "The Matrix", "The Matrix Reloaded", "The Matrix Revolutions"]
```

Rows: 1

List comprehension 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 262. Result

result

```
[0.0, 8.0, 64.0, 216.0, 512.0, 1000.0]
```

Rows: 1

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 263. Result

result

```
[0, 2, 4, 6, 8, 10]
```

Rows: 1

Query

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

Table 264. Result

result

```
[0.0, 1.0, 8.0, 27.0, 64.0, 125.0, 216.0, 343.0, 512.0, 729.0, 1000.0]
```

Rows: 1

## Maps

Cypher supports the construction of maps. This section first discusses literal maps and then moves on to map projection.



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

## Literal maps

The key names in a map must be literals. 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 265. Result

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

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

Rows: 1

## Map projection

Cypher supports map projections, which allows for the construction of 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 projection

```
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:

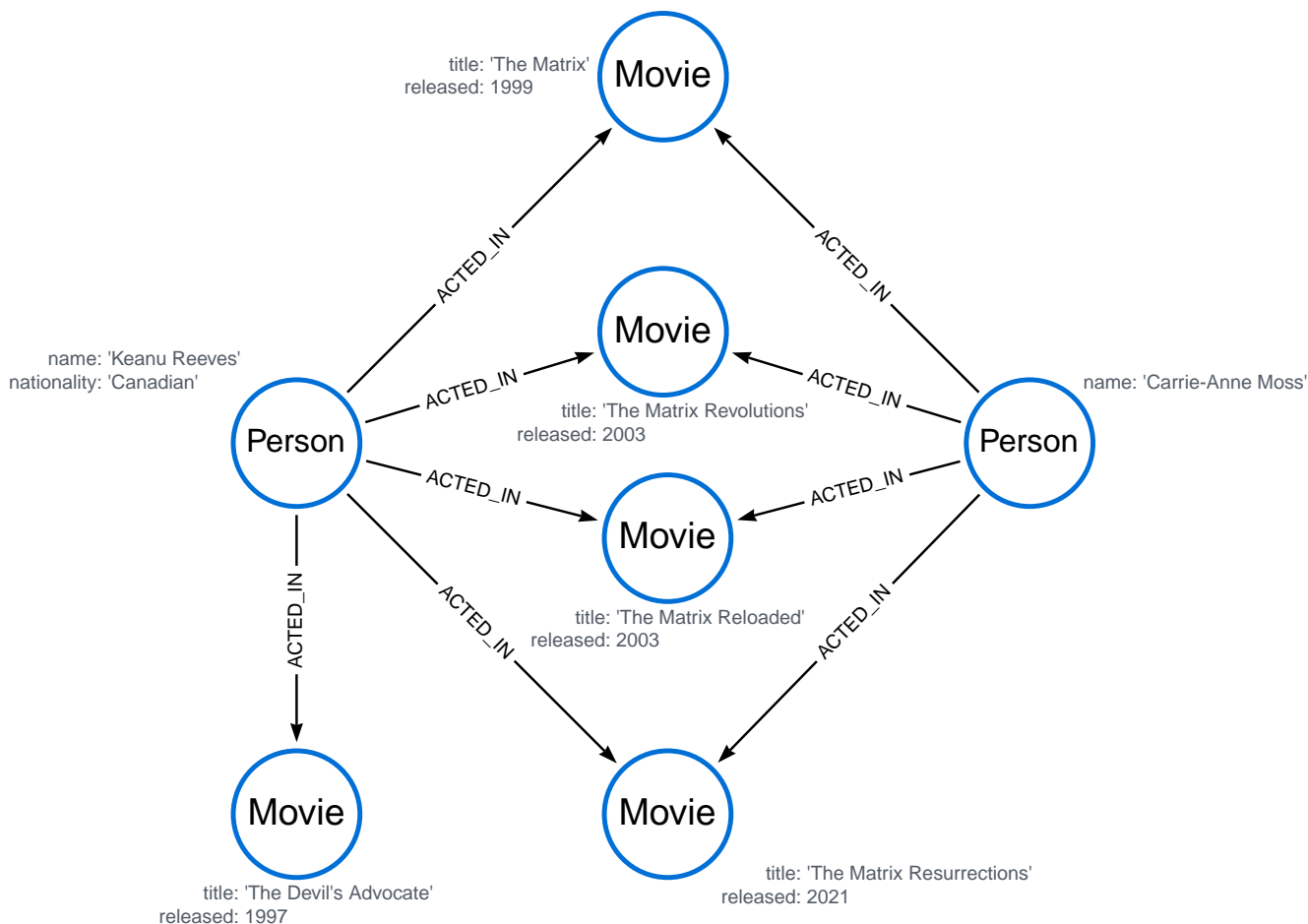
- Property selector - Projects the property name as the key, and the value from the `map_variable` as the value for the projection.
- Literal entry - This is a key-value pair, with the value being an arbitrary expression `key: <expression>`.
- Variable selector - Projects a variable, with the variable 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`.

## Example graph

The following graph is used for the examples below:



To recreate the graph, run the following query against an empty Neo4j database:

```

CREATE
  (keanu:Person {name: 'Keanu Reeves', nationality: 'Canadian'}),
  (carrieAnne:Person {name: 'Carrie-Anne Moss'}),
  (theMatrixRevolutions:Movie {title: 'The Matrix Revolutions', released: 2003}),
  (theMatrixReloaded:Movie {title: 'The Matrix Reloaded', released: 2003}),
  (theMatrix:Movie {title: 'The Matrix', released: 1999}),
  (theDevilsAdvocate:Movie {title: 'The Devils Advocate', released: 1997}),
  (theMatrixResurrections:Movie {title: 'The Matrix Resurrections', released: 2021}),
  (keanu)-[:ACTED_IN]->(theMatrix),
  (keanu)-[:ACTED_IN]->(theMatrixRevolutions),
  (keanu)-[:ACTED_IN]->(theMatrixReloaded),
  (keanu)-[:ACTED_IN]->(theMatrixResurrections),
  (keanu)-[:ACTED_IN]->(theDevilsAdvocate),
  (carrieAnne)-[:ACTED_IN]->(theMatrix),
  (carrieAnne)-[:ACTED_IN]->(theMatrixRevolutions),
  (carrieAnne)-[:ACTED_IN]->(theMatrixReloaded),
  (carrieAnne)-[:ACTED_IN]->(theMatrixResurrections)
  
```

## Examples

The below query finds the **Keanu Reeves** node and the movies he has acted in. It is an example of a map projection with a literal entry, which in turn also uses map projection inside the aggregating `collect()` function.

### Query

```

MATCH (keanu:Person {name: 'Keanu Reeves'})-[:ACTED_IN]->(movie:Movie)
WITH keanu, collect(movie{.title, .released}) AS movies
RETURN keanu{.name, movies: movies}
  
```

Table 266. Result

keanu
{movies: [{title: "The Devils Advocate", released: 1997}, {title: "The Matrix Revolutions", released: 2003}, {title: "The Matrix Resurrections", released: 2021}, {title: "The Matrix Reloaded", released: 2003}, {title: "The Matrix", released: 1999}], name: "Keanu Reeves"}
Rows: 1

The below query finds all **Person** nodes in the graph that have one or more relationships with the type **ACTED\_IN** connected to **Movie** nodes. It uses the **count()** function to count how many **Movie** nodes are connected to each **Person** node in this way, and uses a variable selector to project the value of the count.

## Query

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

Table 267. Result

actor
{numberOfMovies: 5, name: "Keanu Reeves"}
{numberOfMovies: 4, name: "Carrie-Anne Moss"}
Rows: 2

The below query returns all properties from the **Keanu Reeves** node. An all-properties selector is used to project all the node properties, and additionally, explicitly project the property **age**. Since this property does not exist on the node **Keanu Reeves**, a **null** value is projected instead.

## Query

```
MATCH (keanu:Person {name: 'Keanu Reeves'})
RETURN keanu{.*, .age}
```

Table 268. Result

keanu
{nationality: "Canadian", name: "Keanu Reeves", age: null}
Rows: 1

The below query is an example of statically accessing individual map members using the **.** operator:

## Query

```
WITH {age: 58, profession: 'Actor'} as keanuStats
RETURN keanuStats.profession AS profession
```

Table 269. Result

profession
"Actor"

profession

Rows: 1

## Casting data values

Cypher supports a number of functions to cast values to different data types. This section will provide an overview of those functions, as well examples of how to use them in practice.

### Functions for converting data values

The following functions are available for casting data values:

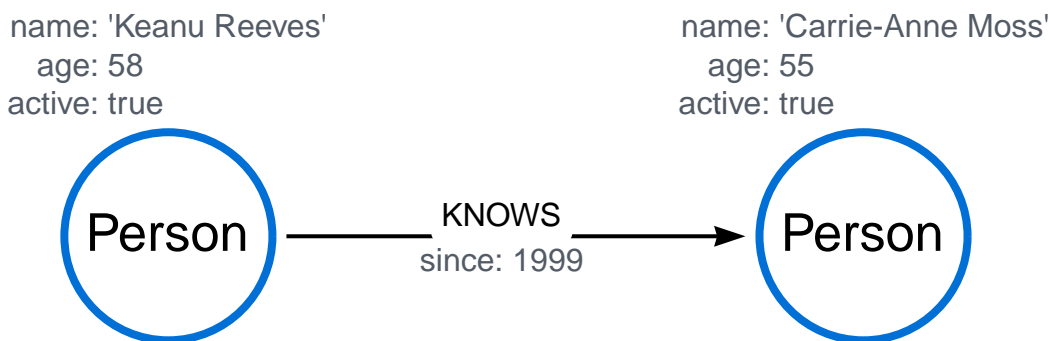
Function	Description
<code>toBoolean()</code>	Converts a string, integer, or boolean value to a boolean value.
<code>toBooleanList()</code>	Converts a list of values and returns a list of boolean values. If any values are not convertible to boolean they will be <code>null</code> in the list returned.
<code>toBooleanOrNull()</code>	Converts a string, integer or boolean value to a boolean value. For any other input value, <code>null</code> will be returned.
<code>toFloat()</code>	Converts an integer, floating point, or a string value to a floating point number value. Otherwise <code>null</code> is returned.
<code>toFloatList()</code>	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 <code>null</code> in the list returned.
<code>toFloatOrNull()</code>	Converts an integer, floating point, or a string value to a floating point number. For any other input value, <code>null</code> will be returned.
<code>toInteger()</code>	Converts a boolean, integer, floating point or a string value to an integer value.
<code>toIntegerList()</code>	Converts a list of values and returns a list of integer values. If any values are not convertible to integer they will be <code>null</code> in the list returned.
<code>toIntegerOrNull()</code>	Converts a boolean, integer, floating point or a string value to an integer value. For any other input value, <code>null</code> will be returned.

Function	Description
<code>toString()</code>	Converts an integer, float, boolean, string, point, duration, date, time, localtime, localdatetime, or datetime value to a string value.
<code>toStringList()</code>	Converts a list of values and returns a list of string values. If any values are not convertible to string they will be <code>null</code> in the list returned.
<code>toStringOrNull()</code>	Converts an integer, float, boolean, string, point, duration, date, time, localtime, localdatetime, or datetime value to a string. For any other input value, <code>null</code> will be returned.

More information about these, and many other functions, can be found in the section on [Functions](#).

## Examples

The following graph is used for the examples below:



To recreate it, run the following query against an empty Neo4j database:

```
CREATE (keanu:Person {name:'Keanu Reeves', age: 58, active:true}),
(carrieAnne:Person {name:'Carrie-Anne Moss', age: 55, active:true}),
(keanu)-[r:KNOWS {since:1999}]->(carrieAnne)
```

## Returning converted values

In the below query, the function `toFloat` is used to cast two string values. It shows that `null` is returned if the data casting is not possible.

```
MATCH (keanu:Person {name:'Keanu Reeves'})
RETURN toFloat(keanu.age), toInteger(keanu.name)
```

Table 270. Result

<code>toFloat(keanu.age)</code>	<code>toInteger(keanu.name)</code>
58.0	null

If the function `toFloat` is passed an unsupported value (such as a `Date`), it will throw an error:

Query

```
WITH date({
  year: 2023, month: 5, day: 2
}) AS d
RETURN toFloat(d)
```

```
Type mismatch: expected Float, Integer, Number or String but was Date (line 4, column 16 (offset: 66))
"RETURN toFloat(d)"
```

However, if the same value is passed to the function `toFloatOrNull`, `null` will be returned.

Query

```
WITH date({
  year: 2023, month: 5, day: 2
}) AS d
RETURN toFloatOrNull(d)
```

Table 271. Result

toFloatOrNull(d)
null

It is also possible to return casted values as a list. The below query uses the `toStringList` to cast all passed values into strings, and return them in a list:

```
MATCH (keanu:Person {name:'Keanu Reeves'})
RETURN toStringList([keanu.name, keanu.age]) AS keanuList
```

Table 272. Result

keanuList
["Keanu Reeves", "58"]

## Updating property value types

The functions to cast data values can be used to update property values on nodes and relationships. The below query casts the `age` (integer), `active` (boolean), and `since` (integer) properties to string values:

```
MATCH (keanu:Person {name:'Keanu Reeves'})-[r:KNOWS]-()
SET keanu.age = toString(keanu.age),
    keanu.active = toString(keanu.active),
    r.since = toString(r.since)
RETURN keanu.age, keanu.active, r.since
```

Table 273. Result

keanu.age	keanu.active	r.since
"58"	"true"	"1999"

[3] The minimum value represents the minimum positive value of a `Float`, i.e. the closest value to zero. It is also possible to have a negative `Float`.

[4] 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.

[5] The [first week of any year](#) is the week that contains the first Thursday of the year, and thus always contains January 4.

[6] 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.

[7] 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).

[8] The expression `datetime().epochMillis` returns the equivalent value of the `timestamp()` function.

[9] For the nanosecond part of the epoch offset, the regular nanosecond component (`instant.nanosecond`) can be used.



# 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](#).



- 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 30. 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.

Function	Signature	Description
<code>all()</code>	<code>all(variable :: VARIABLE IN list :: LIST OF ANY? WHERE predicate :: ANY?) :: (BOOLEAN?)</code>	Returns true if the predicate holds for all elements in the given list.

Function	Signature	Description
<code>any()</code>	<code>any(variable :: VARIABLE IN list :: LIST OF ANY? WHERE predicate :: ANY?) :: (BOOLEAN?)</code>	Returns true if the predicate holds for at least one element in the given list.
<code>exists()</code>	<code>exists(input :: ANY?) :: (BOOLEAN?)</code>	Returns true if a match for the pattern exists in the graph, or if the specified property exists in the node, relationship or map.
<code>isEmpty()</code>	<code>isEmpty(input :: LIST? OF ANY?) :: (BOOLEAN?)</code>	Checks whether a list is empty.
	<code>isEmpty(input :: MAP?) :: (BOOLEAN?)</code>	Checks whether a map is empty.
	<code>isEmpty(input :: STRING?) :: (BOOLEAN?)</code>	Checks whether a string is empty.
<code>none()</code>	<code>none(variable :: VARIABLE IN list :: LIST OF ANY? WHERE predicate :: ANY?) :: (BOOLEAN?)</code>	Returns true if the predicate holds for no element in the given list.
<code>single()</code>	<code>single(variable :: VARIABLE IN list :: LIST OF ANY? WHERE predicate :: ANY?) :: (BOOLEAN?)</code>	Returns true if the predicate holds for exactly one of the elements in the given list.

## Scalar functions

These functions return a single value.

Function	Signature	Description
<code>coalesce()</code>	<code>coalesce(input :: ANY?) :: (ANY?)</code>	Returns the first non-null value in a list of expressions.
<code>endNode()</code>	<code>endNode(input :: RELATIONSHIP?) :: (NODE?)</code>	Returns the end node of a relationship.
<code>head()</code>	<code>head(list :: LIST? OF ANY?) :: (ANY?)</code>	Returns the first element in a list.
<code>id()</code>	<code>id(input :: NODE?) :: (INTEGER?)</code>	Returns the id of a node.
	<code>id(input :: RELATIONSHIP?) :: (INTEGER?)</code>	Returns the id of a relationship.
<code>last()</code>	<code>last(list :: LIST? OF ANY?) :: (ANY?)</code>	Returns the last element in a list.
<code>length()</code>	<code>length(input :: PATH?) :: (INTEGER?)</code>	Returns the length of a path.
	<code>properties(input :: MAP?) :: (MAP?)</code>	Returns a map containing all the properties of a map.
	<code>properties(input :: NODE?) :: (MAP?)</code>	Returns a map containing all the properties of a node.
<code>properties()</code>	<code>properties(input :: RELATIONSHIP?) :: (MAP?)</code>	Returns a map containing all the properties of a relationship.
	<code>randomUUID()</code>	<code>randomUUID() :: (STRING?)</code>

Function	Signature	Description
size()	size(input :: LIST? OF ANY?) :: (INTEGER?)	Returns the number of items in a list.
	size(input :: STRING?) :: (INTEGER?)	Returns the number of Unicode characters in a string.
startNode()	startNode(input :: RELATIONSHIP?) :: (NODE?)	Returns the start node of a relationship.
toBoolean()	toBoolean(input :: STRING?) :: (BOOLEAN?)	Converts a string value to a boolean value.
	toBoolean(input :: BOOLEAN?) :: (BOOLEAN?)	Converts a boolean value to a boolean value.
	toBoolean(input :: INTEGER?) :: (BOOLEAN?)	Converts an integer value to a boolean value.
toBooleanOrNull()	toBooleanOrNull(input :: ANY?) :: (BOOLEAN?)	Converts a value to a boolean value, or null if the value cannot be converted.
toFloat()	toFloat(input :: NUMBER?) :: (FLOAT?)	Converts a number value to a floating point value.
	toFloat(input :: STRING?) :: (FLOAT?)	Converts a string value to a floating point value.
toFloatOrNull()	toFloatOrNull(input :: ANY?) :: (FLOAT?)	Converts a value to a floating point value, or null if the value cannot be converted.
toInteger()	toInteger(input :: NUMBER?) :: (INTEGER?)	Converts a number value to an integer value.
	toInteger(input :: BOOLEAN?) :: (INTEGER?)	Converts a boolean value to an integer value.
	toInteger(input :: STRING?) :: (INTEGER?)	Converts a string value to an integer value.
toIntegerOrNull()	toIntegerOrNull(input :: ANY?) :: (INTEGER?)	Converts a value to an integer value, or null if the value cannot be converted.
type()	type(input :: RELATIONSHIP?) :: (STRING?)	Returns the string representation of the relationship type.

## Aggregating functions

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

Function	Signature	Description
avg()	avg(input :: DURATION?) :: (DURATION?)	Returns the average of a set of duration values.
	avg(input :: FLOAT?) :: (FLOAT?)	Returns the average of a set of floating point values.
	avg(input :: INTEGER?) :: (INTEGER?)	Returns the average of a set of integer values.
collect()	collect(input :: ANY?) :: (LIST? OF ANY?)	Returns a list containing the values returned by an expression.
count()	count(input :: ANY?) :: (INTEGER?)	Returns the number of values or rows.
max()	max(input :: ANY?) :: (ANY?)	Returns the maximum value in a set of values.
min()	min(input :: ANY?) :: (ANY?)	Returns the minimum value in a set of values.
percentileCont()	percentileCont(input :: FLOAT?, percentile :: FLOAT?) :: (FLOAT?)	Returns the percentile of a value over a group using linear interpolation.
percentileDisc()	percentileDisc(input :: FLOAT?, percentile :: FLOAT?) :: (FLOAT?)	Returns the nearest floating point value to the given percentile over a group using a rounding method.
	percentileDisc(input :: INTEGER?, percentile :: FLOAT?) :: (INTEGER?)	Returns the nearest integer value to the given percentile over a group using a rounding method.
stdev()	stdev(input :: FLOAT?) :: (FLOAT?)	Returns the standard deviation for the given value over a group for a sample of a population.
stdevp()	stdevp(input :: FLOAT?) :: (FLOAT?)	Returns the standard deviation for the given value over a group for an entire population.
sum()	sum(input :: DURATION?) :: (DURATION?)	Returns the sum of a set of durations
	sum(input :: FLOAT?) :: (FLOAT?)	Returns the sum of a set of floats
	sum(input :: INTEGER?) :: (INTEGER?)	Returns the sum of a set of integers

## List functions

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

Function	Signature	Description
keys()	keys(input :: MAP?) :: (LIST? OF STRING?)	Returns a list containing the string representations for all the property names of a map.
	keys(input :: NODE?) :: (LIST? OF STRING?)	Returns a list containing the string representations for all the property names of a node.
	keys(input :: RELATIONSHIP?) :: (LIST? OF STRING?)	Returns a list containing the string representations for all the property names of a relationship.
labels()	labels(input :: NODE?) :: (LIST? OF STRING?)	Returns a list containing the string representations for all the labels of a node.
nodes()	nodes(input :: PATH?) :: (LIST? OF NODE?)	Returns a list containing all the nodes in a path.
range()	range(start :: INTEGER?, end :: INTEGER?) :: (LIST? OF INTEGER?)	Returns a list comprising all integer values within a specified range.
	range(start :: INTEGER?, end :: INTEGER?, step :: INTEGER?) :: (LIST? OF INTEGER?)	Returns a list comprising all integer values within a specified range created with step length.
reduce()	reduce(accumulator :: VARIABLE = initial :: ANY?, variable :: VARIABLE IN list :: LIST OF ANY?   expression :: ANY) :: (ANY?)	Runs an expression against individual elements of a list, storing the result of the expression in an accumulator.
relationships()	relationships(input :: PATH?) :: (LIST? OF RELATIONSHIP?)	Returns a list containing all the relationships in a path.
reverse()	reverse(input :: LIST? OF ANY?) :: (LIST? OF ANY?)	Returns a list in which the order of all elements in the original list have been reversed.
tail()	tail(input :: LIST? OF ANY?) :: (LIST? OF ANY?)	Returns all but the first element in a list.
toBooleanList()	toBooleanList(input :: LIST? OF ANY?) :: (LIST? OF BOOLEAN?)	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.
toFloatList()	toFloatList(input :: LIST? OF ANY?) :: (LIST? OF FLOAT?)	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.

Function	Signature	Description
<code>toIntegerList()</code>	<code>toIntegerList(input :: LIST? OF ANY?) :: (LIST? OF INTEGER?)</code>	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.
<code>toStringList()</code>	<code>toStringList(input :: LIST? OF ANY?) :: (LIST? OF STRING?)</code>	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.

## Numeric functions

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

Function	Signature	Description
<code>abs()</code>	<code>abs(input :: FLOAT?) :: (FLOAT?)</code>	Returns the absolute value of a floating point number.
	<code>abs(input :: INTEGER?) :: (INTEGER?)</code>	Returns the absolute value of an integer.
<code>ceil()</code>	<code>ceil(input :: FLOAT?) :: (FLOAT?)</code>	Returns the smallest floating point number that is greater than or equal to a number and equal to a mathematical integer.
<code>floor()</code>	<code>floor(input :: FLOAT?) :: (FLOAT?)</code>	Returns the largest floating point number that is less than or equal to a number and equal to a mathematical integer.
<code>rand()</code>	<code>rand() :: (FLOAT?)</code>	Returns a random floating point number in the range from 0 (inclusive) to 1 (exclusive); i.e. [0,1).
<code>round()</code>	<code>round(input :: FLOAT?) :: (FLOAT?)</code>	Returns the value of a number rounded to the nearest integer.
	<code>round(value :: FLOAT?, precision :: NUMBER?) :: (FLOAT?)</code>	Returns the value of a number rounded to the specified precision using rounding mode HALF_UP.
	<code>round(value :: FLOAT?, precision :: NUMBER?, mode :: STRING?) :: (FLOAT?)</code>	Returns the value of a number rounded to the specified precision with the specified rounding mode.

Function	Signature	Description
sign()	sign(input :: FLOAT?) :: (INTEGER?)	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.
	sign(input :: INTEGER?) :: (INTEGER?)	Returns the signum of an integer number: 0 if the number is 0, -1 for any negative number, and 1 for any positive number.

## Logarithmic functions

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

Function	Signature	Description
e()	e() :: (FLOAT?)	Returns the base of the natural logarithm, e.
exp()	exp(input :: FLOAT?) :: (FLOAT?)	Returns $e^n$ , where e is the base of the natural logarithm, and n is the value of the argument expression.
log()	log(input :: FLOAT?) :: (FLOAT?)	Returns the natural logarithm of a number.
log10()	log10(input :: FLOAT?) :: (FLOAT?)	Returns the common logarithm (base 10) of a number.
sqrt()	sqrt(input :: FLOAT?) :: (FLOAT?)	Returns the square root of a number.

## 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.

Function	Signature	Description
acos()	acos(input :: FLOAT?) :: (FLOAT?)	Returns the arccosine of a number in radians.
asin()	asin(input :: FLOAT?) :: (FLOAT?)	Returns the arcsine of a number in radians.
atan()	atan(input :: FLOAT?) :: (FLOAT?)	Returns the arctangent of a number in radians.
atan2()	atan2(y :: FLOAT?, x :: FLOAT?) :: (FLOAT?)	Returns the arctangent2 of a set of coordinates in radians.
cos()	cos(input :: FLOAT?) :: (FLOAT?)	Returns the cosine of a number.
cot()	cot(input :: FLOAT?) :: (FLOAT?)	Returns the cotangent of a number.

Function	Signature	Description
<code>degrees()</code>	<code>degrees(input :: FLOAT?) :: (FLOAT?)</code>	Converts radians to degrees.
<code>haversin()</code>	<code>haversin(input :: FLOAT?) :: (FLOAT?)</code>	Returns half the versine of a number.
<code>pi()</code>	<code>pi() :: (FLOAT?)</code>	Returns the mathematical constant pi.
<code>radians()</code>	<code>radians(input :: FLOAT?) :: (FLOAT?)</code>	Converts degrees to radians.
<code>sin()</code>	<code>sin(input :: FLOAT?) :: (FLOAT?)</code>	Returns the sine of a number.
<code>tan()</code>	<code>tan(input :: FLOAT?) :: (FLOAT?)</code>	Returns the tangent of a number.

## String functions

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

Function	Signature	Description
<code>left()</code>	<code>left(original :: STRING?, length :: INTEGER?) :: (STRING?)</code>	Returns a string containing the specified number of leftmost characters of the original string.
<code>ltrim()</code>	<code>ltrim(input :: STRING?) :: (STRING?)</code>	Returns the original string with leading whitespace removed.
<code>replace()</code>	<code>replace(original :: STRING?, search :: STRING?, replace :: STRING?) :: (STRING?)</code>	Returns a string in which all occurrences of a specified search string in the original string have been replaced by another (specified) replace string.
<code>reverse()</code>	<code>reverse(input :: STRING?) :: (STRING?)</code>	Returns a string in which the order of all characters in the original string have been reversed.
<code>right()</code>	<code>right(original :: STRING?, length :: INTEGER?) :: (STRING?)</code>	Returns a string containing the specified number of rightmost characters of the original string.
<code>rtrim()</code>	<code>rtrim(input :: STRING?) :: (STRING?)</code>	Returns the original string with trailing whitespace removed.
<code>split()</code>	<code>split(original :: STRING?, splitDelimiter :: STRING?) :: (LIST? OF STRING?)</code>	Returns a list of strings resulting from the splitting of the original string around matches of the given delimiter.
	<code>split(original :: STRING?, splitDelimiters :: LIST? OF STRING?) :: (LIST? OF STRING?)</code>	Returns a list of strings resulting from the splitting of the original string around matches of any of the given delimiters.
<code>substring()</code>	<code>substring(original :: STRING?, start :: INTEGER?) :: (STRING?)</code>	Returns a substring of the original string, beginning with a 0-based index start.
	<code>substring(original :: STRING?, start :: INTEGER?, length :: INTEGER?) :: (STRING?)</code>	Returns a substring of length 'length' of the original string, beginning with a 0-based index start.



Function	Signature	Description
<code>toLowerCase()</code>	<code>toLowerCase(input :: STRING?) :: (STRING?)</code>	Returns the original string in lowercase.
<code>toString()</code>	<code>toString(input :: ANY?) :: (STRING?)</code>	Converts an integer, float, boolean, point or temporal type (i.e. Date, Time, LocalTime, DateTime, LocalDateTime or Duration) value to a string.
<code>toStringOrNull()</code>	<code>toStringOrNull(input :: ANY?) :: (STRING?)</code>	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.
<code>toUpperCase()</code>	<code>toUpperCase(input :: STRING?) :: (STRING?)</code>	Returns the original string in uppercase.
<code>trim()</code>	<code>trim(input :: STRING?) :: (STRING?)</code>	Returns the original string with leading and trailing whitespace removed.

### Temporal instant types functions

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

Function	Signature	Description
<code>date()</code>	<code>date(input = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (DATE?)</code>	Create a Date instant.
<code>date.realtime()</code>	<code>date.realtime(timezone = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (DATE?)</code>	Get the current Date instant using the realtime clock.
<code>date.statement()</code>	<code>date.statement(timezone = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (DATE?)</code>	Get the current Date instant using the statement clock.
<code>date.transaction()</code>	<code>date.transaction(timezone = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (DATE?)</code>	Get the current Date instant using the transaction clock.
<code>date.truncate()</code>	<code>date.truncate(unit :: STRING?, input = DEFAULT_TEMPORAL_ARGUMENT :: ANY?, fields = null :: MAP?) :: (DATE?)</code>	Truncate the input temporal value to a Date instant using the specified unit.
<code>datetime()</code>	<code>datetime(input = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (DATETIME?)</code>	Create a DateTime instant.
<code>datetime.fromepoch()</code>	<code>datetime.fromepoch(seconds :: NUMBER?, nanoseconds :: NUMBER?) :: (DATETIME?)</code>	Create a DateTime given the seconds and nanoseconds since the start of the epoch.
<code>datetime.fromepochmillis()</code>	<code>datetime.fromepochmillis(millisecond s :: NUMBER?) :: (DATETIME?)</code>	Create a DateTime given the milliseconds since the start of the epoch.
<code>datetime.realtime()</code>	<code>datetime.realtime(timezone = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (DATETIME?)</code>	Get the current DateTime instant using the realtime clock.

Function	Signature	Description
<code>datetime.statement()</code>	<code>datetime.statement(timezone = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (DATETIME?)</code>	Get the current DateTime instant using the statement clock.
<code>datetime.transaction()</code>	<code>datetime.transaction(timezone = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (DATETIME?)</code>	Get the current DateTime instant using the transaction clock.
<code>datetime.truncate()</code>	<code>datetime.truncate(unit :: STRING?, input = DEFAULT_TEMPORAL_ARGUMENT :: ANY?, fields = null :: MAP?) :: (DATETIME?)</code>	Truncate the input temporal value to a DateTime instant using the specified unit.
<code>localdatetime()</code>	<code>localdatetime(input = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (LOCALDATETIME?)</code>	Create a LocalDateTime instant.
<code>localdatetime.realtime()</code>	<code>localdatetime.realtime(timezone = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (LOCALDATETIME?)</code>	Get the current LocalDateTime instant using the realtime clock.
<code>localdatetime.statement()</code>	<code>localdatetime.statement(timezone = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (LOCALDATETIME?)</code>	Get the current LocalDateTime instant using the statement clock.
<code>localdatetime.transaction()</code>	<code>localdatetime.transaction(timezone = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (LOCALDATETIME?)</code>	Get the current LocalDateTime instant using the transaction clock.
<code>localdatetime.truncate()</code>	<code>localdatetime.truncate(unit :: STRING?, input = DEFAULT_TEMPORAL_ARGUMENT :: ANY?, fields = null :: MAP?) :: (LOCALDATETIME?)</code>	Truncate the input temporal value to a LocalDateTime instant using the specified unit.
<code>localtime()</code>	<code>localtime(input = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (LOCALTIME?)</code>	Create a LocalTime instant.
<code>localtime.realtime()</code>	<code>localtime.realtime(timezone = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (LOCALTIME?)</code>	Get the current LocalTime instant using the realtime clock.
<code>localtime.statement()</code>	<code>localtime.statement(timezone = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (LOCALTIME?)</code>	Get the current LocalTime instant using the statement clock.
<code>localtime.transaction()</code>	<code>localtime.transaction(timezone = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (LOCALTIME?)</code>	Get the current LocalTime instant using the transaction clock.
<code>localtime.truncate()</code>	<code>localtime.truncate(unit :: STRING?, input = DEFAULT_TEMPORAL_ARGUMENT :: ANY?, fields = null :: MAP?) :: (LOCALTIME?)</code>	Truncate the input temporal value to a LocalTime instant using the specified unit.
<code>time()</code>	<code>time(input = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (TIME?)</code>	Create a Time instant.
<code>time.realtime()</code>	<code>time.realtime(timezone = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (TIME?)</code>	Get the current Time instant using the realtime clock.
<code>time.statement()</code>	<code>time.statement(timezone = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (TIME?)</code>	Get the current Time instant using the statement clock.

Function	Signature	Description
<code>time.transaction()</code>	<code>time.transaction(timezone = DEFAULT_TEMPORAL_ARGUMENT :: ANY?) :: (TIME?)</code>	Get the current Time instant using the transaction clock.
<code>time.truncate()</code>	<code>time.truncate(unit :: STRING?, input = DEFAULT_TEMPORAL_ARGUMENT :: ANY?, fields = null :: MAP?) :: (TIME?)</code>	Truncate the input temporal value to a Time instant using the specified unit.

## Temporal duration functions

Duration values of the [temporal types](#) can be created manipulated using the following functions:

Function	Signature	Description
<code>duration()</code>	<code>duration(input :: ANY?) :: (DURATION?)</code>	Construct a Duration value.
<code>duration.between()</code>	<code>duration.between(from :: ANY?, to :: ANY?) :: (DURATION?)</code>	Compute the duration between the 'from' instant (inclusive) and the 'to' instant (exclusive) in logical units.
<code>duration.inDays()</code>	<code>duration.inDays(from :: ANY?, to :: ANY?) :: (DURATION?)</code>	Compute the duration between the 'from' instant (inclusive) and the 'to' instant (exclusive) in days.
<code>duration.inMonths()</code>	<code>duration.inMonths(from :: ANY?, to :: ANY?) :: (DURATION?)</code>	Compute the duration between the 'from' instant (inclusive) and the 'to' instant (exclusive) in months.
<code>duration.inSeconds()</code>	<code>duration.inSeconds(from :: ANY?, to :: ANY?) :: (DURATION?)</code>	Compute the duration between the 'from' instant (inclusive) and the 'to' instant (exclusive) in seconds.

## Spatial functions

These functions are used to specify 2D or 3D points in a geographic or cartesian Coordinate Reference System and to calculate the geodesic distance between two points.

Function	Signature	Description
<code>point.distance()</code>	<code>point.distance(from :: POINT?, to :: POINT?) :: (FLOAT?)</code>	Returns a floating point number representing the geodesic distance between any two points in the same CRS.
<code>point() - Cartesian 2D</code>	<code>point(input :: MAP?) :: (POINT?)</code>	Returns a 2D point object, given two coordinate values in the Cartesian coordinate system.
<code>point() - Cartesian 3D</code>	<code>point(input :: MAP?) :: (POINT?)</code>	Returns a 3D point object, given three coordinate values in the Cartesian coordinate system.
<code>point() - WGS 84 2D</code>	<code>point(input :: MAP?) :: (POINT?)</code>	Returns a 2D point object, given two coordinate values in the WGS 84 geographic coordinate system.

Function	Signature	Description
<code>point() - WGS 84 3D</code>	<code>point(input :: MAP?) :: (POINT?)</code>	Returns a 3D point object, given three coordinate values in the WGS 84 geographic coordinate system.
<code>point.withinBBox()</code>	<code>point.withinBBox(point :: POINT?, lowerLeft :: POINT?, upperRight :: POINT?) :: (BOOLEAN?)</code>	Returns <code>true</code> if the provided point is within the bounding box defined by the two provided points, <code>lowerLeft</code> and <code>upperRight</code> .

## LOAD CSV functions

LOAD CSV functions can be used to get information about the file that is processed by `LOAD CSV`.

Function	Signature	Description
<code>file()</code>	<code>file() :: (STRING?)</code>	Returns the absolute path of the file that <code>LOAD CSV</code> is using.
<code>linenumber()</code>	<code>linenumber() :: (INTEGER?)</code>	Returns the line number that <code>LOAD CSV</code> is currently using.

## 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:

Type	Description	Usage	Developing
Scalar	For each row the function takes parameters and returns a result.	<a href="#">Using UDF</a>	<a href="#">Extending Neo4j (UDF)</a>
Aggregating	Consumes many rows and produces an aggregated result.	<a href="#">Using aggregating UDF</a>	<a href="#">Extending Neo4j (Aggregating UDF)</a>

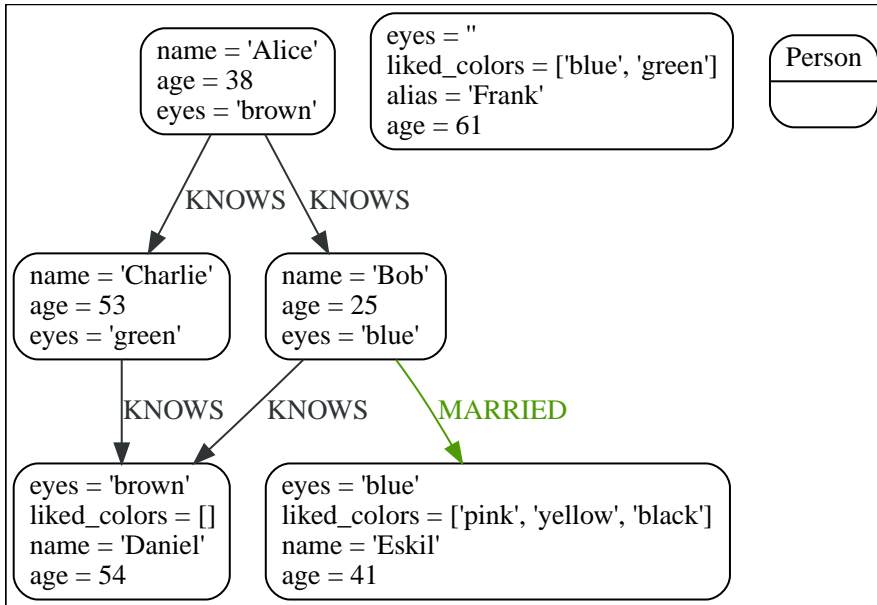
## Predicate functions

*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()`



## `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:

Name	Description
<code>list</code>	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.
<code>variable</code>	A variable that can be used from within the predicate.
<code>predicate</code>	A predicate that is tested against all items in the list.

## Example 31. all()

### 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 274. Result

p
[{"name": "Alice", "eyes": "brown", "age": 38}, {}, {"name": "Charlie", "eyes": "green", "age": 53}, {"name": "Charlie", "eyes": "green", "age": 53}, {}, {"name": "Daniel", "eyes": "brown", "age": 54}]
Rows: 1

## 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:

Name	Description
<code>list</code>	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.
<code>variable</code>	A variable that can be used from within the predicate.
<code>predicate</code>	A predicate that is tested against all items in the list.

## Example 32. any()

### 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 275. Result

n
Node[4]{eyes:"blue",liked_colors:["pink","yellow","black"],name:"Eskil",age:41}
Rows: 1

## 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:

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

### Example 33. exists()

#### 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 276. Result

name	is_married
"Alice"	false
"Bob"	true
"Charlie"	false
"Daniel"	false
"Eskil"	false
Rows: 5	

### Example 34. exists()

#### 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 277. Result

a_name	b_name	b_has_name	c_name	c_has_name
"Alice"	<null>	false	<null>	<null>
Rows: 1				





Note that the `exists()` function is deprecated for property input. Please use the `IS NOT NULL predicate` instead.

## 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:

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

### Example 35. isEmpty(list)

#### Query

```
MATCH (n)
WHERE NOT isEmpty(n.liked_colors)
RETURN n
```

The nodes with the property `liked_colors` being non-empty are returned.

#### Table 278. Result

n
<code>Node[4]{eyes:"blue",liked_colors:["pink","yellow","black"],name:"Eskil",age:41}</code>
<code>Node[5]{eyes:"",liked_colors:["blue","green"],alias:"Frank",age:61}</code>

Rows: 2

### Syntax:

```
isEmpty(map)
```

### Returns:

A Boolean.

## Arguments:

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

## Example 36. isEmpty(map)

### Query

```
MATCH (n)
WHERE isEmpty(properties(n))
RETURN n
```

Nodes that does not have any properties are returned.

### Table 279. Result

n
<code>Node[6]{} </code>
Rows: 1

## Syntax:

```
isEmpty(string)
```

## Returns:

A Boolean.

## Arguments:

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

### Example 37. isEmpty(string)

#### 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 280. Result

age
61
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.

## 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:

Name	Description
<code>list</code>	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.
<code>variable</code>	A variable that can be used from within the predicate.
<code>predicate</code>	A predicate that is tested against all items in the list.

### Example 38. none()

#### Query

```
MATCH p = (n)-[*1..3]->(b)
WHERE
  n.name = 'Alice'
  AND none(x IN nodes(p) WHERE x.age = 25)
RETURN p
```

No node in the returned paths has a property `age` with the value `25`.

#### Table 281. Result

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

## 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:

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

### Example 39. single()

#### Query

```
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'`.

#### Table 282. Result

p
(0)-[KNOWS,0]->(1)
Rows: 1

## 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()`



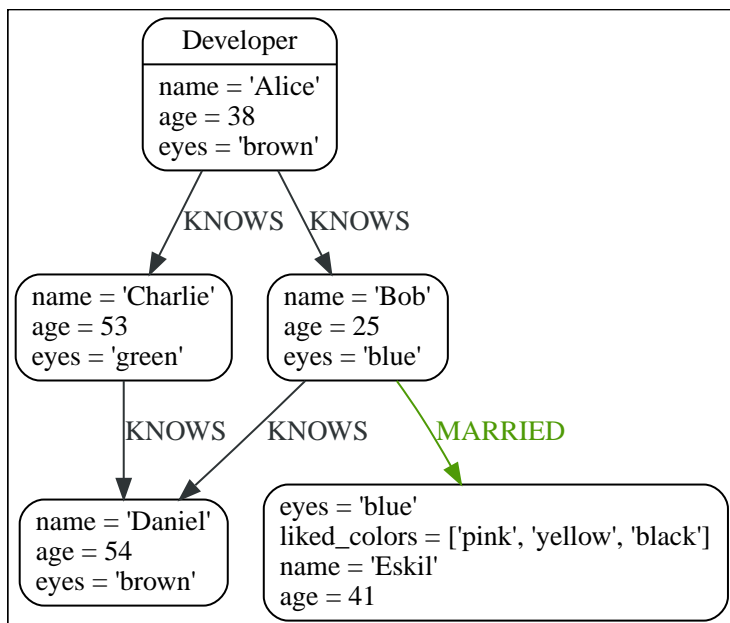
The `length()` and `size()` functions are quite similar, and so it is important to take note of the difference.

Function `length()`

Only works for `paths`.

Function `size()`

Only works for the three types: `strings`, `lists`, `pattern comprehension`.



## `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:

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

Considerations:

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

#### Example 40. `coalesce()`

##### Query

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

##### Table 283. Result

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

## `endNode()`

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

##### Syntax:

```
endNode(relationship)
```

##### Returns:

A Node.

##### Arguments:

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

##### Considerations:

`endNode(null)` returns `null`.

## Example 41. endNode()

### Query

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

### Table 284. Result

endNode(r)
{name: "Bob", age: 25, eyes: "blue"}
{name: "Charlie", age: 53, eyes: "green"}
Rows: 2

## 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:

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



## Example 42. head()

### Query

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

The first element in the list is returned.

Table 285. Result

a.liked_colors	head(a.liked_colors)
["pink", "yellow", "black"]	"pink"
Rows: 1	

## 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.

On a Fabric database, the `id()` function should be used with caution.

When called in database-specific subqueries, the resulting `id` value for a node or relationship is local to that database. The local `id` for nodes or relationships from different databases may be the same.

When called from the root context of a query, the resulting value is an extended `id` for the node or relationship. The extended `id` is likely different from the local `id` for the same node or relationship.

### Syntax:

```
id(expression)
```

### Returns:

An Integer.

### Arguments:

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

### Considerations:

`id(null)` returns `null`.

### Example 43. id()

#### Query

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

The node identifier for each of the nodes is returned.

#### Table 286. Result

id(a)
0
1
2
3
4
Rows: 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:

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

## Considerations:

`last(null)` returns `null`.

`last([])` returns `null`.

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

## Example 44. last()

### Query

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

The last element in the list is returned.

### Table 287. Result

a.liked_colors	last(a.liked_colors)
<code>["pink", "yellow", "black"]</code>	<code>"black"</code>

Rows: 1

## length()

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

## Syntax:

```
length(path)
```

## Returns:

An Integer.

## Arguments:

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

## Considerations:

`length(null)` returns `null`.

#### Example 45. `length()`

##### Query

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

The length of the path `p` is returned.

##### Table 288. Result

<code>length(p)</code>
2
2
2
Rows: 3

## `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:

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

##### Considerations:

`properties(null)` returns `null`.

### Example 46. properties()

#### Query

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

#### Table 289. Result

properties(p)
<pre>{"city": "Berlin", "name": "Stefan"}</pre>

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

## 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.

### Example 47. randomUUID()

#### Query

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

#### Table 290. Result

uuid
<pre>"9f4c297d-309a-4743-a196-4525b96135c1"</pre>

Rows: 1

A randomly-generated UUID is returned.

## size()

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

### Syntax:

```
size(list)
```

### Returns:

An Integer.

### Arguments:

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

### Considerations:

`size(null)` returns `null`.

### Example 48. size()

#### Query

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

#### Table 291. Result

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

2

Rows: 1

The number of elements in the list is returned.

## 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:

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

## Example 49. size()

### Query

```
MATCH (a)
WHERE a.name = 'Alice'
RETURN size([p=(a)-->()->() | p]) AS fof
```

### Table 292. Result

fof
3
Rows: 1

The number of paths matching the pattern expression is returned. (The size of the list of paths).

## size() applied to string

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

### Syntax:

```
size(string)
```

### Returns:

An Integer.

### Arguments:

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

### Considerations:

`size(null)` returns `null`.

## Example 50. size()

### Query

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

### Table 293. Result

size(a.name)
7
Rows: 1

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

## startNode()

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

### Syntax:

```
startNode(relationship)
```

### Returns:

A Node.

### Arguments:

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

### Considerations:

`startNode(null)` returns `null`.



### Example 51. startNode()

#### Query

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

#### Table 294. Result

startNode(r)
{name:"Alice",age:38,eyes:"brown"}
{name:"Alice",age:38,eyes:"brown"}
Rows: 2

## 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.

### Example 52. timestamp()

#### Query

```
RETURN timestamp()
```

The time in milliseconds is returned.

#### Table 295. Result

timestamp()
1655201331965
Rows: 1

## toBoolean()

The function `toBoolean()` converts a string, integer or boolean value to a boolean value.

### Syntax:

```
toBoolean(expression)
```

### Returns:

A Boolean.

### Arguments:

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

### Considerations:

`toBoolean(null)` returns `null`.

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

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

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.

### Example 53. toBoolean()

#### Query

```
RETURN toBoolean('true'), toBoolean('not a boolean'), toBoolean(0)
```

Table 296. Result

<code>toBoolean('true')</code>	<code>toBoolean('not a boolean')</code>	<code>toBoolean(0)</code>
<code>true</code>	<code>&lt;null&gt;</code>	<code>false</code>
Rows: 1		

## 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:

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

### 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.

### Example 54. toBooleanOrNull()

#### Query

```
RETURN toBooleanOrNull('true'), toBooleanOrNull('not a boolean'), toBooleanOrNull(0),  
toBooleanOrNull(1.5)
```

#### Table 297. Result

<code>toBooleanOrNull('true')</code>	<code>toBooleanOrNull('not a boolean')</code>	<code>toBooleanOrNull(0)</code>	<code>toBooleanOrNull(1.5)</code>
<code>true</code>	<code>&lt;null&gt;</code>	<code>false</code>	<code>&lt;null&gt;</code>
Rows: 1			

## 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:

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

## Considerations:

`toFloat(null)` returns `null`.

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

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

This function will return an error if provided with an expression that is not an integer, floating point or a string value.

## Example 55. toFloat()

### Query

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

### Table 298. Result

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

## 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:

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

## 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.

### Example 56. toFloatOrNull()

#### Query

```
RETURN toFloatOrNull('11.5'), toFloatOrNull('not a number'), toFloatOrNull(true)
```

Table 299. Result

<code>toFloatOrNull('11.5')</code>	<code>toFloatOrNull('not a number')</code>	<code>toFloatOrNull(true)</code>
11.5	<null>	<null>

Rows: 1

## 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:

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

#### Considerations:

`toInteger(null)` returns `null`.

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

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

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.

## Example 57. toInteger()

### Query

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

### Table 300. Result

toInteger('42')	toInteger('not a number')	toInteger(true)
42	<null>	1
Rows: 1		

## 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:

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

### 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.

## Example 58. toIntegerOrNull()

### Query

```
RETURN toIntegerOrNull('42'), toIntegerOrNull('not a number'), toIntegerOrNull(true),  
toIntegerOrNull(['A', 'B', 'C'])
```

### Table 301. Result

toIntegerOrNull('42')	toIntegerOrNull('not a number')	toIntegerOrNull(true)	toIntegerOrNull(['A', 'B', 'C'])
42	<null>	1	<null>
Rows: 1			

## type()

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

### Syntax:

```
type(relationship)
```

### Returns:

A String.

### Arguments:

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

### Considerations:

`type(null)` returns `null`.

### Example 59. type()

#### Query

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

The relationship type of `r` is returned.

#### Table 302. Result

type(r)
"KNOWS"
"KNOWS"

Rows: 2

## Aggregating functions

*Aggregating functions take a set of values and calculate an aggregated value over them.*

#### Functions:

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



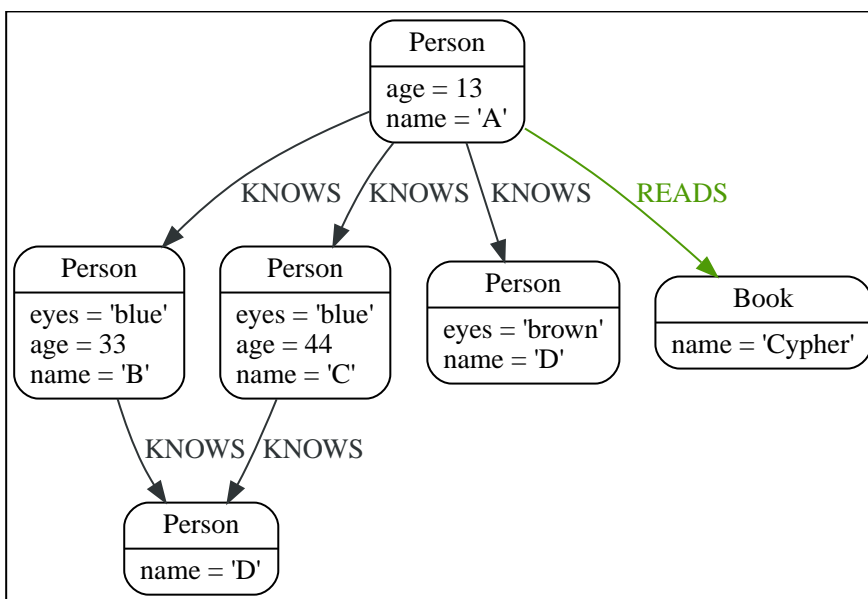
```
RETURN n, count(*)
```

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:



## 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:

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

## Considerations:

Any `null` values are excluded from the calculation.

`avg(null)` returns `null`.

## Example 60. avg()

### Query

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

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

### Table 303. Result

avg(n.age)
30.0

Rows: 1

## avg() - Durations

The function `avg()` returns the average of a set of Durations.

### Syntax:

```
avg(expression)
```

### Returns:

A Duration.

### Arguments:

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

## Considerations:

Any `null` values are excluded from the calculation.

`avg(null)` returns `null`.

## Example 61. avg()

### Query

```
UNWIND [duration('P2DT3H'), duration('PT1H45S')] AS dur
RETURN avg(dur)
```

The average of the two supplied Durations is returned.

### Table 304. Result

avg(dur)
P1DT2H22.5S
Rows: 1

## 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:

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

### Considerations:

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

`collect(null)` returns an empty list.

## Example 62. collect()

### Query

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

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

### Table 305. Result

collect(n.age)
[13,33,44]
Rows: 1

## 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:

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

### Considerations:

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

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

`count(null)` returns 0.



Neo4j maintains a transactional count store for holding count metadata, which can significantly increase the speed of queries using the `count()` function. For more information about the count store, refer to [Neo4j Knowledge Base → Fast counts using the count store](#).

## 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`.

### Example 63. `count()`

#### 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 306. Result

labels(n)	n.age	count(*)
["Person"]	13	4
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.

### Example 64. `count()`

#### Query

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

The type of matched relationships are grouped and the group count are returned.

Table 307. Result

type(r)	count(*)
"KNOWS"	3
"READS"	1
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.

Example 65. `count()`

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 308. Result

count(n.age)
3
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.

Example 66. `count()`

Query

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

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

Table 309. Result

count(DISTINCT friend_of_friend)	count(friend_of_friend)
1	2
Rows: 1	

# 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:

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

## Considerations:

Any `null` values are excluded from the calculation.

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

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

`max(null)` returns `null`.

## Example 67. max()

### 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 310. Result

max(val)
1
Rows: 1

### Example 68. max()

#### 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 311. Result

max(val)
<code>[1,2]</code>
Rows: 1

### Example 69. max()

#### Query

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

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

#### Table 312. Result

max(n.age)
<code>44</code>
Rows: 1

## 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:



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

### Considerations:

Any `null` values are excluded from the calculation.

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

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

`min(null)` returns `null`.

### Example 70. `min()`

#### 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 313. Result

<code>min(val)</code>
<code>"1"</code>
Rows: 1

### Example 71. `min()`

#### Query

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

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

#### Table 314. Result

<code>min(val)</code>
<code>["a", "c", 23]</code>
Rows: 1

## Example 72. min()

### Query

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

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

### Table 315. Result

min(n.age)
13
Rows: 1

## 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:

Name	Description
<code>expression</code>	A numeric expression.
<code>percentile</code>	A numeric value between <code>0.0</code> and <code>1.0</code> .

### Considerations:

Any `null` values are excluded from the calculation.

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

### Example 73. percentileCont()

#### 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 316. Result

percentileCont(n.age, 0.4)
29.0
Rows: 1

## 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:

Name	Description
<code>expression</code>	A numeric expression.
<code>percentile</code>	A numeric value between <code>0.0</code> and <code>1.0</code> .

#### Considerations:

Any `null` values are excluded from the calculation.

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

## Example 74. percentileDisc()

### Query

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

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

### Table 317. Result

percentileDisc(n.age, 0.5)
33
Rows: 1

## 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:

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

### Considerations:

Any `null` values are excluded from the calculation.

`stDev(null)` returns `0`.

## Example 75. stDev()

### Query

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

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

### Table 318. Result

stDev(n.age)
15.716233645501712
Rows: 1

## 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:

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

### Considerations:

Any `null` values are excluded from the calculation.

`stDevP(null)` returns `0`.

## Example 76. stDevP()

### 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 319. Result

stDevP(n.age)
12.832251036613439
Rows: 1

## 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:

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

### Considerations:

Any `null` values are excluded from the calculation.

`sum(null)` returns `0`.

## Example 77. sum()

### Query

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

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

### Table 320. Result

sum(n.age)
90
Rows: 1

## sum() - Durations

The function `sum()` returns the sum of a set of durations.

### Syntax:

```
sum(expression)
```

### Returns:

A Duration.

### Arguments:

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

### Considerations:

Any `null` values are excluded from the calculation.

## Example 78. sum()

### Query

```
UNWIND [duration('P2DT3H'), duration('PT1H45S')] AS dur
RETURN sum(dur)
```

The sum of the two supplied Durations is returned.

### Table 321. Result

sum(dur)
P2DT4H45S

Rows: 1

## List functions

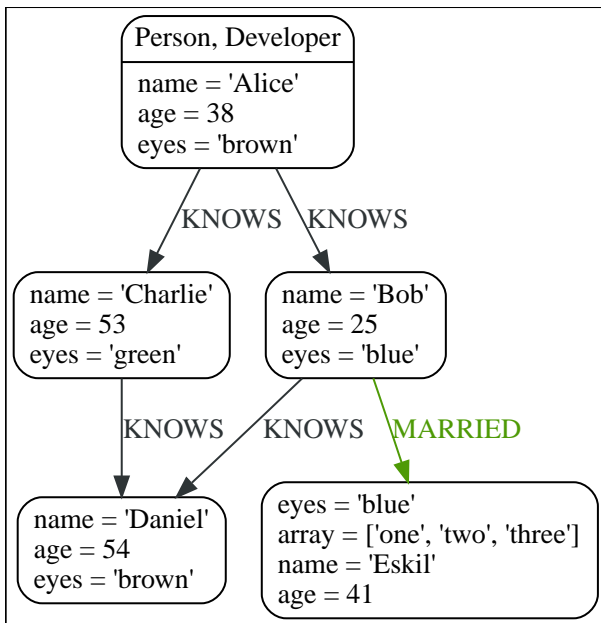
*List functions return lists of things — nodes in a path, and so on.*

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\(\)](#)





## keys()

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

### Syntax:

```
keys(expression)
```

### Returns:

A list containing String elements.

### Arguments:

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

### Considerations:

`keys(null)` returns `null`.

## Example 79. keys()

### 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 322. Result

keys(a)
["name", "age", "eyes"]
Rows: 1

## labels()

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

### Syntax:

```
labels(node)
```

### Returns:

A list containing String elements.

### Arguments:

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

### Considerations:

`labels(null)` returns `null`.

## Example 80. labels()

### Query

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

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

### Table 323. Result

labels(a)
["Person", "Developer"]
Rows: 1

## nodes()

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

### Syntax:

```
nodes(path)
```

### Returns:

A list containing Node elements.

### Arguments:

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

### Considerations:

`nodes(null)` returns `null`.

## Example 81. nodes()

### Query

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

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

### Table 324. Result

nodes(p)
[Node[0]{name:"Alice",age:38,eyes:"brown"},Node[1]{name:"Bob",age:25,eyes:"blue"},Node[4]{eyes:"blue",array:["one","two","three"],name:"Eskil",age:41}]
Rows: 1

## 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:

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

## Example 82. range()

### Query

```
RETURN range(0, 10), range(2, 18, 3), range(0, 5, -1)
```

Three lists of numbers in the given ranges are returned.

Table 325. Result

range(0, 10)	range(2, 18, 3)	range(0, 5, -1)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]	[2, 5, 8, 11, 14, 17]	[]
Rows: 1		

## 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:

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

### Example 83. reduce()

#### 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 326. Result

reduction
117
Rows: 1

## relationships()

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

#### Syntax:

```
relationships(path)
```

#### Returns:

A list containing Relationship elements.

#### Arguments:

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

#### Considerations:

`relationships(null)` returns `null`.

## Example 84. relationships()

### Query

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

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

### Table 327. Result

relationships(p)
<code>[ :KNOWS[0]{} , :MARRIED[4]{} ]</code>
Rows: 1

## reverse()

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

### Syntax:

```
reverse(original)
```

### Returns:

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

### Arguments:

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

### Considerations:

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

### Example 85. reverse()

#### Query

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

#### Table 328. Result

reverse(ids)
[487,<null>,521,"abc",4923]
Rows: 1

## tail()

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

#### Syntax:

```
tail(list)
```

#### Returns:

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

#### Arguments:

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

### Example 86. tail()

#### Query

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

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

#### Table 329. Result

a.array	tail(a.array)
["one", "two", "three"]	["two", "three"]
Rows: 1	



## toBooleanList()

`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:

```
toBooleanList(list)
```

### Returns:

A list containing the converted elements; depending on the input value a converted value is either a boolean value or `null`.

### Arguments:

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

### Considerations:

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

Any boolean 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 `toBooleanOrNull()` function.

### Example 87. toBooleanList()

#### Query

```
RETURN toBooleanList(null) as noList,  
toBooleanList([null, null]) as nullsInList,  
toBooleanList(['a string', true, 'false', null, ['A','B']]) as mixedList
```

Table 330. Result

noList	nullsInList	mixedList
<null>	[<null>,<null>]	[<null>, true, false, <null>, <null>]

Rows: 1

## 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:

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

### 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.

### Example 88. toFloatList()

#### Query

```
RETURN toFloatList(null) as noList,  
toFloatList([null, null]) as nullsInList,  
toFloatList(['a string', 2.5, '3.14159', null, ['A','B']]) as mixedList
```

#### Table 331. Result

noList	nullsInList	mixedList
<code>&lt;null&gt;</code>	<code>[&lt;null&gt;, &lt;null&gt;]</code>	<code>[&lt;null&gt;, 2.5, 3.14159, &lt;null&gt;, &lt;null&gt;]</code>

Rows: 1

## 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:

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

### 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.

### Example 89. toIntegerList()

#### Query

```
RETURN toIntegerList(null) as noList,  
toIntegerList([null, null]) as nullsInList,  
toIntegerList(['a string', 2, '5', null, ['A','B']]) as mixedList
```

#### Table 332. Result

noList	nullsInList	mixedList
<null>	[<null>, <null>]	[<null>, 2, 5, <null>, <null>]

Rows: 1

## 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 `null`.

### Arguments:

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

### Considerations:

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

Any string 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 `toStringOrNull()` function.

### Example 90. toStringList()

#### Query

```
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
```

Table 333. Result

noList	nullsInList	mixedList
<code>&lt;null&gt;</code>	<code>[&lt;null&gt;, &lt;null&gt;]</code>	<code>["already a string", "2", "1955-11-05", &lt;null&gt;, &lt;null&gt;]</code>
Rows: 1		

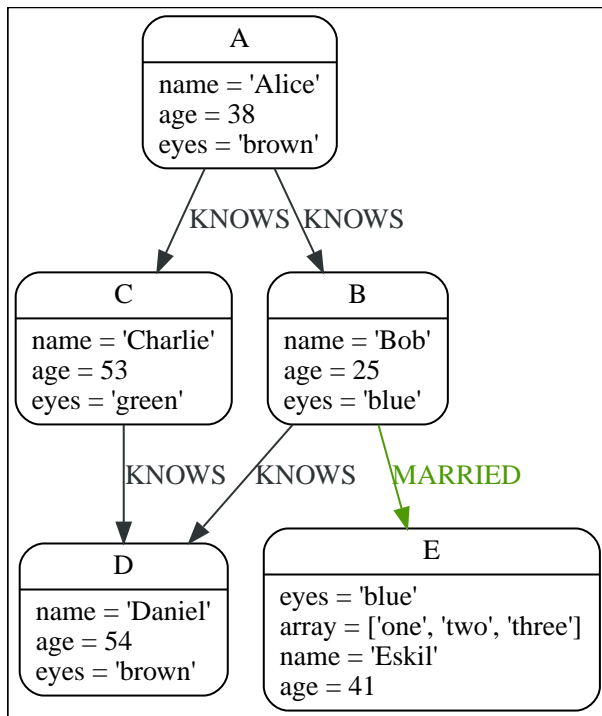
## Mathematical functions - numeric

These functions all operate on numeric expressions only, and will return an error if used on any other values. See also [Mathematical operators](#).

### Functions:

- [abs\(\)](#)
- [ceil\(\)](#)
- [floor\(\)](#)
- [rand\(\)](#)
- [round\(\)](#)
- [round\(\), with precision](#)
- [round\(\), with precision and rounding mode](#)
- [sign\(\)](#)

The following graph is used for the examples below:



abs()

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

Syntax:

```
abs(expression)
```

Returns:

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

Arguments:

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

Considerations:

`abs(null)` returns `null`.

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

## Example 91. abs()

### 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 334. Result

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

## ceil()

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

### Syntax:

```
ceil(expression)
```

### Returns:

A Float.

### Arguments:

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

### Considerations:

`ceil(null)` returns `null`.

## Example 92. ceil()

### Query

```
RETURN ceil(0.1)
```

The ceil of 0.1 is returned.

Table 335. Result

ceil(0.1)
1.0
Rows: 1

## floor()

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

### Syntax:

```
floor(expression)
```

### Returns:

A Float.

### Arguments:

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

### Considerations:

`floor(null)` returns `null`.

### Example 93. floor()

#### Query

```
RETURN floor(0.9)
```

The floor of 0.9 is returned.

#### Table 336. Result

floor(0.9)
0.0

Rows: 1

## 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.

### Example 94. rand()

#### Query

```
RETURN rand()
```

A random number is returned.

#### Table 337. Result

rand()
0.5460251846326871

Rows: 1

## round()

`round()` returns the value of the given number rounded to the nearest integer, with ties always rounded towards positive infinity.



## Syntax:

```
round(expression)
```

## Returns:

A Float.

## Arguments:

Name	Description
<code>expression</code>	A numeric expression to be rounded.

## Considerations:

`round(null)` returns `null`.

## Example 95. round()

### Query

```
RETURN round(3.141592)
```

`3.0` is returned.

### Table 338. Result

```
round(3.141592)
```

`3.0`

Rows: 1

## Example 96. round() of negative number with tie

### Query

```
RETURN round(-1.5)
```

Ties are rounded towards positive infinity, therefore `-1.0` is returned.

### Table 339. Result

```
round(-1.5)
```

`-1.0`

Rows: 1

## round(), with precision

`round()` returns the value of the given number rounded to the closest value of given precision, with ties always being rounded away from zero (using rounding mode `HALF_UP`). The exception is for precision 0, where ties are rounded towards positive infinity to align with `round()` without precision.

### Syntax:

```
round(expression, precision)
```

### Returns:

A Float.

### Arguments:

Name	Description
<code>expression</code>	A numeric expression to be rounded.
<code>precision</code>	A numeric expression specifying precision.

### Considerations:

`round()` returns `null` if any of its input parameters are `null`.

### Example 97. round() with precision

#### Query

```
RETURN round(3.141592, 3)
```

3.142 is returned.

#### Table 340. Result

```
round(3.141592, 3)
```

3.142

Rows: 1

### Example 98. round() with precision 0 and tie

#### Query

```
RETURN round(-1.5, 0)
```

To align with `round(-1.5)`, `-1.0` is returned.

#### Table 341. Result

<code>round(-1.5, 0)</code>
<code>-1.0</code>
Rows: 1

### Example 99. round() with precision 1 and tie

#### Query

```
RETURN round(-1.55, 1)
```

The default is to round away from zero when there is a tie, therefore `-1.6` is returned.

#### Table 342. Result

<code>round(-1.55, 1)</code>
<code>-1.6</code>
Rows: 1

## 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:

Name	Description
<code>expression</code>	A numeric expression to be rounded.

Name	Description
precision	A numeric expression specifying precision.
mode	A string expression specifying rounding mode.

### Modes:

Mode	Description
UP	Round away from zero.
DOWN	Round towards zero.
CEILING	Round towards positive infinity.
FLOOR	Round towards negative infinity.
HALF_UP	Round towards closest value of given precision, with ties always being rounded away from zero.
HALF_DOWN	Round towards closest value of given precision, with ties always being rounded towards zero.
HALF_EVEN	Round towards closest value of given precision, with ties always being rounded to the even neighbor.

### Considerations:

For the rounding modes, a tie means that the two closest values of the given precision are at the same distance from the given value. E.g. for precision 1, 2.15 is a tie as it has equal distance to 2.1 and 2.2, while 2.151 is not a tie, as it is closer to 2.2.

`round()` returns `null` if any of its input parameters are `null`.

### Example 100. `round()` with precision and UP rounding mode

#### Query

```
RETURN round(1.249, 1, 'UP') AS positive,
round(-1.251, 1, 'UP') AS negative,
round(1.25, 1, 'UP') AS positiveTie,
round(-1.35, 1, 'UP') AS negativeTie
```

The rounded values using precision 1 and rounding mode `UP` are returned.

Table 343. Result

positive	negative	positiveTie	negativeTie
1.3	-1.3	1.3	-1.4

Rows: 1

Example 101. round() with precision and DOWN rounding mode

Query

```
RETURN round(1.249, 1, 'DOWN') AS positive,  
round(-1.251, 1, 'DOWN') AS negative,  
round(1.25, 1, 'DOWN') AS positiveTie,  
round(-1.35, 1, 'DOWN') AS negativeTie
```

The rounded values using precision 1 and rounding mode **DOWN** are returned.

Table 344. Result

positive	negative	positiveTie	negativeTie
1.2	-1.2	1.2	-1.3
Rows: 1			

Example 102. round() with precision and CEILING rounding mode

Query

```
RETURN round(1.249, 1, 'CEILING') AS positive,  
round(-1.251, 1, 'CEILING') AS negative,  
round(1.25, 1, 'CEILING') AS positiveTie,  
round(-1.35, 1, 'CEILING') AS negativeTie
```

The rounded values using precision 1 and rounding mode **CEILING** are returned.

Table 345. Result

positive	negative	positiveTie	negativeTie
1.3	-1.2	1.3	-1.3
Rows: 1			

Example 103. round() with precision and FLOOR rounding mode

Query

```
RETURN round(1.249, 1, 'FLOOR') AS positive,  
round(-1.251, 1, 'FLOOR') AS negative,  
round(1.25, 1, 'FLOOR') AS positiveTie,  
round(-1.35, 1, 'FLOOR') AS negativeTie
```

The rounded values using precision 1 and rounding mode **FLOOR** are returned.

Table 346. Result

positive	negative	positiveTie	negativeTie
1.2	-1.3	1.2	-1.4
Rows: 1			

Example 104. round() with precision and HALF\_UP rounding mode

Query

```
RETURN round(1.249, 1, 'HALF_UP') AS positive,  
round(-1.251, 1, 'HALF_UP') AS negative,  
round(1.25, 1, 'HALF_UP') AS positiveTie,  
round(-1.35, 1, 'HALF_UP') AS negativeTie
```

The rounded values using precision 1 and rounding mode HALF\_UP are returned.

Table 347. Result

positive	negative	positiveTie	negativeTie
1.2	-1.3	1.3	-1.4
Rows: 1			

Example 105. round() with precision and HALF\_DOWN rounding mode

Query

```
RETURN round(1.249, 1, 'HALF_DOWN') AS positive,  
round(-1.251, 1, 'HALF_DOWN') AS negative,  
round(1.25, 1, 'HALF_DOWN') AS positiveTie,  
round(-1.35, 1, 'HALF_DOWN') AS negativeTie
```

The rounded values using precision 1 and rounding mode HALF\_DOWN are returned.

Table 348. Result

positive	negative	positiveTie	negativeTie
1.2	-1.3	1.2	-1.3
Rows: 1			

Example 106. round() with precision and HALF\_EVEN rounding mode

Query

```
RETURN round(1.249, 1, 'HALF_EVEN') AS positive,  
round(-1.251, 1, 'HALF_EVEN') AS negative,  
round(1.25, 1, 'HALF_EVEN') AS positiveTie,  
round(-1.35, 1, 'HALF_EVEN') AS negativeTie
```

The rounded values using precision 1 and rounding mode HALF\_EVEN are returned.

Table 349. Result

positive	negative	positiveTie	negativeTie
1.2	-1.3	1.2	-1.4
Rows: 1			

## sign()

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

### Syntax:

```
sign(expression)
```

### Returns:

An Integer.

### Arguments:

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

### Considerations:

`sign(null)` returns `null`.

### Example 107. sign()

#### Query

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

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

Table 350. Result

sign(-17)	sign(0.1)
<code>-1</code>	<code>1</code>
Rows: 1	

## 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()`

## `e()`

`e()` returns the base of the natural logarithm, *e*.

### Syntax:

```
e()
```

### Returns:

A Float.

### Example 108. `e()`

#### Query

```
RETURN e()
```

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

#### Table 351. Result

<code>e()</code>
2.718281828459045
Rows: 1

## `exp()`

`exp()` returns  $e^n$ , where *e* is the base of the natural logarithm, and *n* is the value of the argument expression.

### Syntax:

```
e(expression)
```

### Returns:

A Float.

### Arguments:



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

### Considerations:

`exp(null)` returns `null`.

`exp()` returns `Infinity` when the return value is greater than the largest `FLOAT` value (Java Double.MAX\_VALUE).

### Example 109. exp()

#### Query

```
RETURN exp(2)
```

`e` to the power of `2` is returned.

#### Table 352. Result

<code>exp(2)</code>
<code>7.38905609893065</code>
Rows: 1

## log()

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

### Syntax:

```
log(expression)
```

### Returns:

A Float.

### Arguments:

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

### Considerations:

`log(null)` returns `null`.

`log(0)` returns `-Infinity`.

If (`expression < 0`), then (`log(expression)`) returns `NaN`.

## Example 110. log()

### Query

```
RETURN log(27)
```

The natural logarithm of 27 is returned.

### Table 353. Result

log(27)
3.295836866004329
Rows: 1

## log10()

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

### Syntax:

```
log10(expression)
```

### Returns:

A Float.

### Arguments:

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

### Considerations:

`log10(null)` returns `null`.

`log10(0)` returns `-Infinity`.

If (`expression < 0`), then (`log10(expression)`) returns `NaN`.

## Example 111. log10()

### Query

```
RETURN log10(27)
```

The common logarithm of 27 is returned.

### Table 354. Result

log10(27)
1.4313637641589874
Rows: 1

## sqrt()

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

### Syntax:

```
sqrt(expression)
```

### Returns:

A Float.

### Arguments:

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

### Considerations:

`sqrt(null)` returns `null`.

If (`expression < 0`), then (`sqrt(expression)`) returns `NaN`.

### Example 112. sqrt()

#### Query

```
RETURN sqrt(256)
```

The square root of 256 is returned.

Table 355. Result

sqrt(256)
16.0
Rows: 1

## 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\(\)](#)

### acos()

[acos\(\)](#) returns the arccosine of a number in radians.

#### Syntax:

```
acos(expression)
```

### Returns:

A Float.

### Arguments:

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

### Considerations:

`acos(null)` returns `null`.

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

### Example 113. acos()

#### Query

```
RETURN acos(0.5)
```

The arccosine of `0.5` is returned.

#### Table 356. Result

```
acos(0.5)
```

```
1.0471975511965979
```

```
Rows: 1
```

## asin()

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

### Syntax:

```
asin(expression)
```

### Returns:

A Float.

### Arguments:

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

**Considerations:**

`asin(null)` returns `null`.

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

**Example 114. asin()**

Query

```
RETURN asin(0.5)
```

The arcsine of `0.5` is returned.

Table 357. Result

<code>asin(0.5)</code>
<code>0.5235987755982989</code>
Rows: 1

## atan()

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

**Syntax:**

```
atan(expression)
```

**Returns:**

A Float.

**Arguments:**

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

**Considerations:**

`atan(null)` returns `null`.

## Example 115. atan()

### Query

```
RETURN atan(0.5)
```

The arctangent of 0.5 is returned.

### Table 358. Result

atan(0.5)
0.4636476090008061
Rows: 1

## atan2()

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

### Syntax:

```
atan2(expression1, expression2)
```

### Returns:

A Float.

### Arguments:

Name	Description
<code>expression1</code>	A numeric expression for y that represents the angle in radians.
<code>expression2</code>	A numeric expression for x that represents the angle in radians.

### Considerations:

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

## Example 116. atan2()

### Query

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

The arctangent2 of 0.5 and 0.6 is returned.

### Table 359. Result

atan2(0.5, 0.6)
0.6947382761967033
Rows: 1

## cos()

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

### Syntax:

```
cos(expression)
```

### Returns:

A Float.

### Arguments:

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

### Considerations:

`cos(null)` returns `null`.



## Example 117. cos()

### Query

```
RETURN cos(0.5)
```

The cosine of 0.5 is returned.

### Table 360. Result

cos(0.5)
0.8775825618903728
Rows: 1

## cot()

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

### Syntax:

```
cot(expression)
```

### Returns:

A Float.

### Arguments:

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

### Considerations:

`cot(null)` returns `null`.

`cot(0)` returns `Infinity`.

## Example 118. cot()

### Query

```
RETURN cot(0.5)
```

The cotangent of 0.5 is returned.

### Table 361. Result

cot(0.5)
1.830487721712452

Rows: 1

## degrees()

`degrees()` converts radians to degrees.

### Syntax:

```
degrees(expression)
```

### Returns:

A Float.

### Arguments:

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

### Considerations:

`degrees(null)` returns `null`.

## Example 119. degrees

### Query

```
RETURN degrees(3.14159)
```

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

### Table 362. Result

degrees(3.14159)
179.9998479605043
Rows: 1

## haversin()

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

### Syntax:

```
haversin(expression)
```

### Returns:

A Float.

### Arguments:

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

### Considerations:

`haversin(null)` returns `null`.

### Example 120. haversin()

#### Query

```
RETURN haversin(0.5)
```

The haversine of 0.5 is returned.

#### Table 363. Result

haversin(0.5)
0.06120871905481362
Rows: 1

## 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).

### Example 121. haversin()

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 364. Result

dist
9129.969740051658
Rows: 1 Nodes created: 2 Properties set: 4 Labels added: 2

## pi()

`pi()` returns the mathematical constant pi.

Syntax:

```
pi()
```

**Returns:**

A Float.

**Example 122. pi()****Query**

```
RETURN pi()
```

The constant pi is returned.

**Table 365. Result**

pi()
3.141592653589793
Rows: 1

## radians()

`radians()` converts degrees to radians.

**Syntax:**

```
radians(expression)
```

**Returns:**

A Float.

**Arguments:**

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

**Considerations:**

`radians(null)` returns `null`.

## Example 123. radians()

### Query

```
RETURN radians(180)
```

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

### Table 366. Result

radians(180)
3.141592653589793
Rows: 1

## sin()

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

### Syntax:

```
sin(expression)
```

### Returns:

A Float.

### Arguments:

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

### Considerations:

`sin(null)` returns `null`.

### Example 124. sin()

#### Query

```
RETURN sin(0.5)
```

The sine of 0.5 is returned.

#### Table 367. Result

sin(0.5)
0.479425538604203

Rows: 1

## tan()

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

#### Syntax:

```
tan(expression)
```

#### Returns:

A Float.

#### Arguments:

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

#### Considerations:

`tan(null)` returns `null`.

## Example 125. tan()

### Query

```
RETURN tan(0.5)
```

The tangent of 0.5 is returned.

Table 368. Result

tan(0.5)
0.5463024898437905
Rows: 1

## 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\(\)](#)
- [toLowerCase\(\)](#)



- `toString()`
- `toStringOrNull()`
- `toUpper()`
- `trim()`

## left()

`left()` returns a string containing the specified number of leftmost characters of the original string.

### Syntax:

```
left(original, length)
```

### Returns:

A String.

### Arguments:

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

### Considerations:

`left(null, length)` return `null`.

`left(null, null)` 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.

### Example 126. left()

#### Query

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

#### Table 369. Result

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

```
"hel"
```

```
Rows: 1
```

## ltrim()

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

### Syntax:

```
ltrim(original)
```

### Returns:

A String.

### Arguments:

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

### Considerations:

`ltrim(null)` returns `null`.

### Example 127. ltrim()

#### Query

```
RETURN ltrim(' hello')
```

#### Table 370. Result

```
ltrim(' hello')
```

```
"hello"
```

```
Rows: 1
```

## replace()

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

### Syntax:

```
replace(original, search, replace)
```

### Returns:

A String.

## Arguments:

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

## Considerations:

If any argument is `null`, `null` will be returned.

If `search` is not found in `original`, `original` will be returned.

## Example 128. `replace()`

### Query

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

### Table 371. Result

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

```
"hewwo"
```

```
Rows: 1
```

## `reverse()`

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

## Syntax:

```
reverse(original)
```

## Returns:

A String.

## Arguments:

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

## Considerations:

`reverse(null)` returns `null`.

## Example 129. reverse

### Query

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

### Table 372. Result

reverse('anagram')
"margana"
Rows: 1

## right()

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

### Syntax:

```
right(original, length)
```

### Returns:

A String.

### Arguments:

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

### Considerations:

`right(null, length)` return `null`.

`right(null, null)` 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.

### Example 130. right()

#### Query

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

#### Table 373. Result

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

```
"llo"
```

```
Rows: 1
```

## rtrim()

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

#### Syntax:

```
rtrim(original)
```

#### Returns:

A String.

#### Arguments:

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

#### Considerations:

`rtrim(null)` returns `null`.

### Example 131. rtrim()

#### Query

```
RETURN rtrim('hello ')
```

#### Table 374. Result

```
rtrim('hello ')
```

```
"hello"
```

```
Rows: 1
```

## split()

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

### Syntax:

```
split(original, splitDelimiter)
```

### Returns:

A list of Strings.

### Arguments:

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

### Considerations:

```
split(null, splitDelimiter) return null.
```

```
split(original, null) return null
```

### Example 132. split()

#### Query

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

#### Table 375. Result

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

```
["one", "two"]
```

Rows: 1

## substring()

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

### Syntax:

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

### Returns:

A String.

### Arguments:

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

### Considerations:

`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.

### Example 133. `substring()`

#### Query

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

#### Table 376. Result

<code>substring('hello', 1, 3)</code>	<code>substring('hello', 2)</code>
"ell"	"llo"
Rows: 1	

## toLowerCase()

`toLowerCase()` returns the original string in lowercase.

### Syntax:

```
toLowerCase(original)
```

### Returns:

A String.

### Arguments:

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

### Considerations:

`toLowerCase(null)` returns `null`.

### Example 134. toLower()

#### Query

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

#### Table 377. Result

```
toLowerCase('HELLO')
```

```
"hello"
```

```
Rows: 1
```

## 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:

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

### Considerations:

`toString(null)` returns `null`.



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, string, boolean, point, duration, date, time, localtime, localdatetime or datetime value.

### Example 135. toString()

#### 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 378. Result

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

Rows: 1

## 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:

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

#### 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.

### Example 136. toStringOrNull()

#### Query

```
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 379. Result

toStringOrNull(11.5)	toStringOrNull('already a string')	toStringOrNull(true)	dateString	datetimeString	durationString	list
"11.5"	"already a string"	"true"	"1984-10-11"	"1984-10-11T12:31:14.341+01:00[Europe/Stockholm]"	"PT11M"	<null>

Rows: 1

## toUpper()

**toUpper()** returns the original string in uppercase.

#### Syntax:

```
toUpper(original)
```

#### Returns:

A String.

#### Arguments:

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

#### Considerations:

**toUpper(null)** returns **null**.

### Example 137. toUpper()

#### Query

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

#### Table 380. Result

toUpper('hello')
"HELLO"
Rows: 1

## trim()

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

#### Syntax:

```
trim(original)
```

#### Returns:

A String.

#### Arguments:

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

#### Considerations:

`trim(null)` returns `null`.

### Example 138. trim()

#### Query

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

#### Table 381. Result

trim(' hello ')
"hello"
Rows: 1

# 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](#).

## Temporal instant types

### 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 382. Temporal instant type creation functions

Function	Date	Time	LocalTime	DateTime	LocalDateTime
Getting the current value.	X	X	X	X	X
Creating a calendar-based (Year-Month-Day) value.	X			X	X

Function	Date	Time	LocalTime	DateTime	LocalDateTime
Creating a week-based (Year-Week-Day) value.	X			X	X
Creating a quarter-based (Year-Quarter-Day) value.	X			X	X
Creating an ordinal (Year-Day) value.	X			X	X
Creating a value from time components.		X	X		
Creating a value from other temporal values using extractors (i.e. converting between different types).	X	X	X	X	X
Creating a value from a string.	X	X	X	X	X
Creating a value from a timestamp.				X	



All the temporal instant types — including those that do not contain time zone information support such as *Date*, *LocalTime* and *DateTime* — allow for a time zone to specified for the functions that retrieve the current instant. This allows for the retrieval of the current instant in the specified time zone.

## Controlling which clock to use

The functions which create temporal instant values based on the current instant use the **statement** clock as default. However, there are three different clocks available for more fine-grained control:

- **transaction**: The same instant is produced for each invocation within the same transaction. A different time may be produced for different transactions.
- **statement**: The same instant is produced for each invocation within the same statement. A different time may be produced for different statements within the same transaction.
- **realtime**: The instant produced will be the live clock of the system.

The following table lists the different sub-functions for specifying the clock to be used when creating the current temporal instant value:

Type	default	transaction	statement	realtime
Date	<code>date()</code>	<code>date.transaction()</code>	<code>date.statement()</code>	<code>date.realtime()</code>
Time	<code>time()</code>	<code>time.transaction()</code>	<code>time.statement()</code>	<code>time.realtime()</code>
LocalTime	<code>localtime()</code>	<code>localtime.transaction()</code>	<code>localtime.statement()</code>	<code>localtime.realtime()</code>
DateTime	<code>datetime()</code>	<code>datetime.transaction()</code>	<code>datetime.statement()</code>	<code>datetime.realtime()</code>
LocalDateTime	<code>localdatetime()</code>	<code>localdatetime.transaction()</code>	<code>localdatetime.statement()</code>	<code>localdatetime.realtime()</code>

## Truncating temporal values

A temporal instant value can be created by truncating another temporal instant value at the nearest preceding point in time at a specified component boundary (namely, a *truncation unit*). A temporal instant value created in this way will have all components which are less significant than the specified truncation unit set to their default values.

It is possible to supplement the truncated value by providing a map containing components which are less significant than the truncation unit. This will have the effect of overriding the default values which would otherwise have been set for these less significant components.

The following truncation units are supported:

- **millennium**: Select the temporal instant corresponding to the *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:

Truncation unit	Date	Time	LocalTime	DateTime	LocalDateTime
millennium	<code>date.truncate('millennium', input)</code>			<code>datetime.truncate('millennium', input)</code>	<code>localdatetime.truncate('millennium', input)</code>
century	<code>date.truncate('century', input)</code>			<code>datetime.truncate('century', input)</code>	<code>localdatetime.truncate('century', input)</code>
decade	<code>date.truncate('decade', input)</code>			<code>datetime.truncate('decade', input)</code>	<code>localdatetime.truncate('decade', input)</code>
year	<code>date.truncate('year', input)</code>			<code>datetime.truncate('year', input)</code>	<code>localdatetime.truncate('year', input)</code>
weekYear	<code>date.truncate('weekYear', input)</code>			<code>datetime.truncate('weekYear', input)</code>	<code>localdatetime.truncate('weekYear', input)</code>
quarter	<code>date.truncate('quarter', input)</code>			<code>datetime.truncate('quarter', input)</code>	<code>localdatetime.truncate('quarter', input)</code>
month	<code>date.truncate('month', input)</code>			<code>datetime.truncate('month', input)</code>	<code>localdatetime.truncate('month', input)</code>
week	<code>date.truncate('week', input)</code>			<code>datetime.truncate('week', input)</code>	<code>localdatetime.truncate('week', input)</code>
day	<code>date.truncate('day', input)</code>	<code>time.truncate('day', input)</code>	<code>localtime.truncate('day', input)</code>	<code>datetime.truncate('day', input)</code>	<code>localdatetime.truncate('day', input)</code>
hour		<code>time.truncate('hour', input)</code>	<code>localtime.truncate('hour', input)</code>	<code>datetime.truncate('hour', input)</code>	<code>localdatetime.truncate('hour', input)</code>
minute		<code>time.truncate('minute', input)</code>	<code>localtime.truncate('minute', input)</code>	<code>datetime.truncate('minute', input)</code>	<code>localdatetime.truncate('minute', input)</code>
second		<code>time.truncate('second', input)</code>	<code>localtime.truncate('second', input)</code>	<code>datetime.truncate('second', input)</code>	<code>localdatetime.truncate('second', input)</code>
millisecond		<code>time.truncate('millisecond', input)</code>	<code>localtime.truncate('millisecond', input)</code>	<code>datetime.truncate('millisecond', input)</code>	<code>localdatetime.truncate('millisecond', input)</code>
microsecond		<code>time.truncate('microsecond', input)</code>	<code>localtime.truncate('microsecond', input)</code>	<code>datetime.truncate('microsecond', input)</code>	<code>localdatetime.truncate('microsecond', input)</code>

## date()

Details for using the `date()` function.

- Getting the current Date
  - `date.transaction()`
  - `date.statement()`
  - `date.realtime()`

- [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:

Name	Description
A single map consisting of the following:	
<code>timezone</code>	A string expression that represents the <a href="#">time zone</a> .

### Considerations:

If no parameters are provided, `date()` must be invoked (`date({})` is invalid).

### Example 139. `date()`

#### Query

```
RETURN date() AS currentDate
```

The current date is returned.

#### Table 383. Result

currentDate
2022-06-14
Rows: 1



## Example 140. date()

### Query

```
RETURN date({timezone: 'America/Los Angeles'}) AS currentDateInLA
```

The current date in California is returned.

### Table 384. Result

currentDateInLA
2022-06-14
Rows: 1

## date.transaction()

`date.transaction()` returns the current Date value using the `transaction` clock. This value will be the same for each invocation within the same transaction. However, a different value may be produced for different transactions.

### Syntax:

```
date.transaction([[timezone]])
```

### Returns:

A Date.

### Arguments:

Name	Description
<code>timezone</code>	A string expression that represents the <a href="#">time zone</a> .

## Example 141. date.transaction()

### Query

```
RETURN date.transaction() AS currentDate
```

### Table 385. Result

currentDate
2022-06-14
Rows: 1

date.statement()

`date.statement()` returns the current Date value using the `statement` clock. This value will be the same for each invocation within the same statement. However, a different value may be produced for different statements within the same transaction.

**Syntax:**

```
date.statement([[{timezone}]])
```

**Returns:**

A Date.

**Arguments:**

Name	Description
<code>timezone</code>	A string expression that represents the <code>time zone</code> .

**Example 142. date.statement()**

Query

```
RETURN date.statement() AS currentDate
```

Table 386. Result

currentDate
2022-06-14

Rows: 1

date.realtime()

`date.realtime()` returns the current Date value using the `realtime` clock. This value will be the live clock of the system.

**Syntax:**

```
date.realtime([[{timezone}]])
```

**Returns:**

A Date.

**Arguments:**

Name	Description
timezone	A string expression that represents the <a href="#">time zone</a> .

#### Example 143. date.realtime()

##### Query

```
RETURN date.realtime() AS currentDate
```

##### Table 387. Result

currentDate
2022-06-14
Rows: 1

#### Example 144. date.realtime()

##### Query

```
RETURN date.realtime('America/Los Angeles') AS currentDateInLA
```

##### Table 388. Result

currentDateInLA
2022-06-14
Rows: 1

## Creating a calendar (Year-Month-Day) Date

`date()` returns a Date value with the specified year, month and day component values.

### Syntax:

```
date({year [, month, day]})
```

### Returns:

A Date.

### Arguments:

Name	Description
A single map consisting of the following:	
year	An expression consisting of at <a href="#">least four digits</a> that specifies the year.

Name	Description
month	An integer between 1 and 12 that specifies the month.
day	An integer between 1 and 31 that specifies the day of the month.

### 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.

### Example 145. date()

#### Query

```
UNWIND [
  date({year: 1984, month: 10, day: 11}),
  date({year: 1984, month: 10}),
  date({year: 1984})
] AS theDate
RETURN theDate
```

#### Table 389. Result

theDate
1984-10-11
1984-10-01
1984-01-01
Rows: 3

## Creating a week (Year-Week-Day) Date

date() returns a Date value with the specified year, week and dayOfWeek component values.

### Syntax:

```
date({year [, week, dayOfWeek]})
```

### Returns:

A Date.

### Arguments:

Name	Description
A single map consisting of the following:	

Name	Description
year	An expression consisting of at least four digits that specifies the year.
week	An integer between 1 and 53 that specifies the week.
dayOfWeek	An integer between 1 and 7 that specifies the day of the week.

### Considerations:

The day of the week component will default to 1 if dayOfWeek is omitted.

The week component will default to 1 if week is omitted.

If week is omitted, dayOfWeek must also be omitted.

### Example 146. date()

#### Query

```
UNWIND [
  date({year: 1984, week: 10, dayOfWeek: 3}),
  date({year: 1984, week: 10}),
  date({year: 1984})
] AS theDate
RETURN theDate
```

#### Table 390. Result

theDate
1984-03-07
1984-03-05
1984-01-01
Rows: 3

## Creating a quarter (Year-Quarter-Day) Date

date() returns a Date value with the specified year, quarter and dayOfQuarter component values.

### Syntax:

```
date({year [, quarter, dayOfQuarter]})
```

### Returns:

A Date.

### Arguments:

Name	Description
A single map consisting of the following:	
<code>year</code>	An expression consisting of at least four digits that specifies the year.
<code>quarter</code>	An integer between 1 and 4 that specifies the quarter.
<code>dayOfQuarter</code>	An integer between 1 and 92 that specifies the day of the quarter.

### Considerations:

The day of the quarter component will default to 1 if `dayOfQuarter` is omitted.

The quarter component will default to 1 if `quarter` is omitted.

If `quarter` is omitted, `dayOfQuarter` must also be omitted.

### Example 147. `date()`

#### Query

```
UNWIND [
  date({year: 1984, quarter: 3, dayOfQuarter: 45}),
  date({year: 1984, quarter: 3}),
  date({year: 1984})
] AS theDate
RETURN theDate
```

#### Table 391. Result

theDate
1984-08-14
1984-07-01
1984-01-01
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:

Name	Description
A single map consisting of the following:	
year	An expression consisting of at least four digits that specifies the year.
ordinalDay	An integer between 1 and 366 that specifies the ordinal day of the year.

### Considerations:

The ordinal day of the year component will default to 1 if ordinalDay is omitted.

### Example 148. date()

#### Query

```
UNWIND [
  date({year: 1984, ordinalDay: 202}),
  date({year: 1984})
] AS theDate
RETURN theDate
```

The date corresponding to 11 February 1984 is returned.

#### Table 392. Result

theDate
1984-07-20
1984-01-01

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:

Name	Description
temporalValue	A string representing a temporal value.

## 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`.

## Example 149. date()

### Query

```
UNWIND [  
  date('2015-07-21'),  
  date('2015-07'),  
  date('201507'),  
  date('2015-W30-2'),  
  date('2015202'),  
  date('2015')  
] AS theDate  
RETURN theDate
```

### Table 393. Result

theDate
2015-07-21
2015-07-01
2015-07-01
2015-07-21
2015-07-21
2015-01-01
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:



Name	Description
A single map consisting of the following:	
date	A Date value.
year	An expression consisting of at least four digits that specifies the year.
month	An integer between 1 and 12 that specifies the month.
day	An integer between 1 and 31 that specifies the day of the month.
week	An integer between 1 and 53 that specifies the week.
dayOfWeek	An integer between 1 and 7 that specifies the day of the week.
quarter	An integer between 1 and 4 that specifies the quarter.
dayOfQuarter	An integer between 1 and 92 that specifies the day of the quarter.
ordinalDay	An integer between 1 and 366 that specifies the ordinal day of the year.

### Considerations:

If any of the optional parameters are provided, these will override the corresponding components of `date`.

`date(dd)` may be written instead of `date({date: dd})`.

### Example 150. date()

#### 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 394. Result

dateOnly	dateDay
1984-11-11	1984-11-28
1984-11-11	1984-11-28
1984-11-11	1984-11-28
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 string is `'year'` in order to ensure the returned value has the day set to `x` instead of the default day (which is `1`).

### Syntax:

```
date.truncate(unit [, temporalInstantValue [, mapOfComponents ] ])
```

### Returns:

A Date.

### Arguments:

Name	Description
<code>unit</code>	A string expression evaluating to one of the following strings: <code>'millennium'</code> , <code>'century'</code> , <code>'decade'</code> , <code>'year'</code> , <code>'weekYear'</code> , <code>'quarter'</code> , <code>'month'</code> , <code>'week'</code> , <code>'day'</code> .
<code>temporalInstantValue</code>	An expression of one of the following types: <code>DateTime</code> , <code>LocalDateTime</code> , <code>Date</code> .
<code>mapOfComponents</code>	An expression evaluating to a map containing components less significant than <code>unit</code> .

### Considerations:

Any component that is provided in `mapOfComponents` must be less significant than `unit`; i.e. if `unit` string 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())`.

## Example 151. date.truncate()

### Query

```
WITH
  datetime({
    year: 2017, month: 11, day: 11,
    hour: 12, minute: 31, second: 14, nanosecond: 645876123,
    timezone: '+01:00'
  }) AS d
RETURN
  date.truncate('millennium', d) AS truncMillenium,
  date.truncate('century', d) AS truncCentury,
  date.truncate('decade', d) AS truncDecade,
  date.truncate('year', d, {day: 5}) AS truncYear,
  date.truncate('weekYear', d) AS truncWeekYear,
  date.truncate('quarter', d) AS truncQuarter,
  date.truncate('month', d) AS truncMonth,
  date.truncate('week', d, {dayOfWeek: 2}) AS truncWeek,
  date.truncate('day', d) AS truncDay
```

Table 395. Result

truncMilleni um	truncCentu ry	truncDecad e	truncYear	truncWeek Year	truncQuart er	truncMonth	truncWeek	truncDay
2000-01-01	2000-01-01	2010-01-01	2017-01-05	2017-01-02	2017-10-01	2017-11-01	2017-11-07	2017-11-11
Rows: 1								

## datetime()

Details for using the `datetime()` function.

- [Getting the current \*DateTime\*](#)
  - `datetime.transaction()`
  - `datetime.statement()`
  - `datetime.realtime()`
- [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:

Name	Description
A single map consisting of the following:	
<code>timezone</code>	A string expression that represents the <a href="#">time zone</a> .

### Considerations:

If no parameters are provided, `datetime()` must be invoked (`datetime({})` is invalid).

### Example 152. `.datetime()`

#### Query

```
RETURN datetime() AS currentDateTime
```

The current date and time using the local time zone is returned.

#### Table 396. Result

currentDateTime
2022-06-14T10:02:28.192Z

Rows: 1

### Example 153. .datetime()

#### Query

```
RETURN datetime({timezone: 'America/Los Angeles'}) AS currentDateTimeInLA
```

The current date and time of day in California is returned.

Table 397. Result

currentDateTimeInLA
2022-06-14T03:02:28.238-07:00[America/Los_Angeles]
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:

Name	Description
<code>timezone</code>	A string expression that represents the <code>time zone</code> .

### Example 154. datetime.transaction()

#### Query

```
RETURN datetime.transaction() AS currentDateTime
```

Table 398. Result

currentDateTime
2022-06-14T10:02:28.290Z
Rows: 1

### Example 155. `datetime.transaction()`

#### Query

```
RETURN datetime.transaction('America/Los Angeles') AS currentDateTimeInLA
```

#### Table 399. Result

currentDateTimeInLA
2022-06-14T03:02:28.338-07:00[America/Los_Angeles]
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:

Name	Description
<code>timezone</code>	A string expression that represents the <a href="#">time zone</a> .

### Example 156. `datetime.statement()`

#### Query

```
RETURN datetime.statement() AS currentDateTime
```

#### Table 400. Result

currentDateTime
2022-06-14T10:02:28.395Z
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:**

Name	Description
<code>timezone</code>	A string expression that represents the <code>time zone</code> .

**Example 157. `datetime.realtime()`**

Query

```
RETURN datetime.realtime() AS currentDateTime
```

Table 401. Result

`currentDateTime`

`2022-06-14T10:02:28.494444Z`

Rows: 1

## Creating a calendar (Year-Month-Day) `DateTime`

`datetime()` returns a `DateTime` value with the specified year, month, day, hour, minute, second, millisecond, microsecond, nanosecond and `timezone` component values.

**Syntax:**

```
datetime({year [, month, day, hour, minute, second, millisecond, microsecond, nanosecond, timezone]})
```

**Returns:**

A `DateTime`.

**Arguments:**

Name	Description
A single map consisting of the following:	
<code>year</code>	An expression consisting of at least four digits that specifies the year.
<code>month</code>	An integer between 1 and 12 that specifies the month.
<code>day</code>	An integer between 1 and 31 that specifies the day of the month.
<code>hour</code>	An integer between 0 and 23 that specifies the hour of the day.
<code>minute</code>	An integer between 0 and 59 that specifies the number of minutes.
<code>second</code>	An integer between 0 and 59 that specifies the number of seconds.
<code>millisecond</code>	An integer between 0 and 999 that specifies the number of milliseconds.
<code>microsecond</code>	An integer between 0 and 999,999 that specifies the number of microseconds.
<code>nanosecond</code>	An integer between 0 and 999,999,999 that specifies the number of nanoseconds.
<code>timezone</code>	An expression that specifies the time zone.

### Considerations:

The month component will default to 1 if <code>month</code> is omitted.
The day of the month component will default to 1 if <code>day</code> is omitted.
The hour component will default to 0 if <code>hour</code> is omitted.
The minute component will default to 0 if <code>minute</code> is omitted.
The second component will default to 0 if <code>second</code> is omitted.
Any missing <code>millisecond</code> , <code>microsecond</code> or <code>nanosecond</code> values will default to 0.
The <code>timezone</code> component will default to the configured default time zone if <code>timezone</code> is omitted.
If <code>millisecond</code> , <code>microsecond</code> and <code>nanosecond</code> are given in combination (as part of the same set of parameters), the individual values must be in the range 0 to 999.
The least significant components in the set <code>year</code> , <code>month</code> , <code>day</code> , <code>hour</code> , <code>minute</code> , and <code>second</code> may be omitted; i.e. it is possible to specify only <code>year</code> , <code>month</code> and <code>day</code> , but specifying <code>year</code> , <code>month</code> , <code>day</code> and <code>minute</code> is not permitted.
One or more of <code>millisecond</code> , <code>microsecond</code> and <code>nanosecond</code> can only be specified as long as <code>second</code> is also specified.



## Example 158. datetime()

### Query

```
UNWIND [
  datetime({year: 1984, month: 10, day: 11, hour: 12, minute: 31, second: 14, millisecond: 123,
    microsecond: 456, nanosecond: 789}),
  datetime({year: 1984, month: 10, day: 11, hour: 12, minute: 31, second: 14, millisecond: 645,
    timezone: '+01:00'}),
  datetime({year: 1984, month: 10, day: 11, hour: 12, minute: 31, second: 14, nanosecond: 645876123,
    timezone: 'Europe/Stockholm'}),
  datetime({year: 1984, month: 10, day: 11, hour: 12, minute: 31, second: 14, timezone: '+01:00'}),
  datetime({year: 1984, month: 10, day: 11, hour: 12, minute: 31, second: 14}),
  datetime({year: 1984, month: 10, day: 11, hour: 12, minute: 31, timezone: 'Europe/Stockholm'}),
  datetime({year: 1984, month: 10, day: 11, hour: 12, timezone: '+01:00'}),
  datetime({year: 1984, month: 10, day: 11, timezone: 'Europe/Stockholm'})
] AS theDate
RETURN theDate
```

### Table 402. Result

theDate
1984-10-11T12:31:14.123456789Z
1984-10-11T12:31:14.645+01:00
1984-10-11T12:31:14.645876123+01:00[Europe/Stockholm]
1984-10-11T12:31:14+01:00
1984-10-11T12:31:14Z
1984-10-11T12:31+01:00[Europe/Stockholm]
1984-10-11T12:00+01:00
1984-10-11T00:00+01:00[Europe/Stockholm]
Rows: 8

## Creating a week (Year-Week-Day) DateTime

`datetime()` returns a `DateTime` value with the specified year, week, `dayOfWeek`, hour, minute, second, millisecond, microsecond, nanosecond and timezone component values.

### Syntax:

```
datetime({year [, week, dayOfWeek, hour, minute, second, millisecond, microsecond, nanosecond, timezone]})
```

### Returns:

A `DateTime`.

### Arguments:

Name	Description
A single map consisting of the following:	

Name	Description
<code>year</code>	An expression consisting of at least four digits that specifies the year.
<code>week</code>	An integer between 1 and 53 that specifies the week.
<code>dayOfWeek</code>	An integer between 1 and 7 that specifies the day of the week.
<code>hour</code>	An integer between 0 and 23 that specifies the hour of the day.
<code>minute</code>	An integer between 0 and 59 that specifies the number of minutes.
<code>second</code>	An integer between 0 and 59 that specifies the number of seconds.
<code>millisecond</code>	An integer between 0 and 999 that specifies the number of milliseconds.
<code>microsecond</code>	An integer between 0 and 999,999 that specifies the number of microseconds.
<code>nanosecond</code>	An integer between 0 and 999,999,999 that specifies the number of nanoseconds.
<code>timezone</code>	An expression that specifies the time zone.

### Considerations:

The <code>week</code> component will default to 1 if <code>week</code> is omitted.
The <code>day of the week</code> component will default to 1 if <code>dayOfWeek</code> is omitted.
The <code>hour</code> component will default to 0 if <code>hour</code> is omitted.
The <code>minute</code> component will default to 0 if <code>minute</code> is omitted.
The <code>second</code> component will default to 0 if <code>second</code> is omitted.
Any missing <code>millisecond</code> , <code>microsecond</code> or <code>nanosecond</code> values will default to 0.
The <code>timezone</code> component will default to the configured default time zone if <code>timezone</code> is omitted.
If <code>millisecond</code> , <code>microsecond</code> and <code>nanosecond</code> are given in combination (as part of the same set of parameters), the individual values must be in the range 0 to 999.
The least significant components in the set <code>year</code> , <code>week</code> , <code>dayOfWeek</code> , <code>hour</code> , <code>minute</code> , and <code>second</code> may be omitted; i.e. it is possible to specify only <code>year</code> , <code>week</code> and <code>dayOfWeek</code> , but specifying <code>year</code> , <code>week</code> , <code>dayOfWeek</code> and <code>minute</code> is not permitted.
One or more of <code>millisecond</code> , <code>microsecond</code> and <code>nanosecond</code> can only be specified as long as <code>second</code> is also specified.

## Example 159. datetime()

### Query

```
UNWIND [
  datetime({year: 1984, week: 10, dayOfWeek: 3, hour: 12, minute: 31, second: 14, millisecond: 645}),
  datetime({year: 1984, week: 10, dayOfWeek: 3, hour: 12, minute: 31, second: 14, microsecond: 645876,
  timezone: '+01:00'}),
  datetime({year: 1984, week: 10, dayOfWeek: 3, hour: 12, minute: 31, second: 14, nanosecond:
  645876123, timezone: 'Europe/Stockholm'}),
  datetime({year: 1984, week: 10, dayOfWeek: 3, hour: 12, minute: 31, second: 14, timezone:
  'Europe/Stockholm'}),
  datetime({year: 1984, week: 10, dayOfWeek: 3, hour: 12, minute: 31, second: 14}),
  datetime({year: 1984, week: 10, dayOfWeek: 3, hour: 12, timezone: '+01:00'}),
  datetime({year: 1984, week: 10, dayOfWeek: 3, timezone: 'Europe/Stockholm'})
] AS theDate
RETURN theDate
```

### Table 403. Result

theDate
1984-03-07T12:31:14.645Z
1984-03-07T12:31:14.645876+01:00
1984-03-07T12:31:14.645876123+01:00[Europe/Stockholm]
1984-03-07T12:31:14+01:00[Europe/Stockholm]
1984-03-07T12:31:14Z
1984-03-07T12:00+01:00
1984-03-07T00:00+01:00[Europe/Stockholm]
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:

Name	Description
A single map consisting of the following:	
<code>year</code>	An expression consisting of at least four digits that specifies the year.

Name	Description
<code>quarter</code>	An integer between 1 and 4 that specifies the quarter.
<code>dayOfQuarter</code>	An integer between 1 and 92 that specifies the day of the quarter.
<code>hour</code>	An integer between 0 and 23 that specifies the hour of the day.
<code>minute</code>	An integer between 0 and 59 that specifies the number of minutes.
<code>second</code>	An integer between 0 and 59 that specifies the number of seconds.
<code>millisecond</code>	An integer between 0 and 999 that specifies the number of milliseconds.
<code>microsecond</code>	An integer between 0 and 999,999 that specifies the number of microseconds.
<code>nanosecond</code>	An integer between 0 and 999,999,999 that specifies the number of nanoseconds.
<code>timezone</code>	An expression that specifies the time zone.

### 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.

## Example 160. datetime()

### Query

```
UNWIND [
  datetime({year: 1984, quarter: 3, dayOfQuarter: 45, hour: 12, minute: 31, second: 14, microsecond:
  645876}),
  datetime({year: 1984, quarter: 3, dayOfQuarter: 45, hour: 12, minute: 31, second: 14, timezone:
  '+01:00'}),
  datetime({year: 1984, quarter: 3, dayOfQuarter: 45, hour: 12, timezone: 'Europe/Stockholm'}),
  datetime({year: 1984, quarter: 3, dayOfQuarter: 45})
] AS theDate
RETURN theDate
```

### Table 404. Result

theDate
1984-08-14T12:31:14.645876Z
1984-08-14T12:31:14+01:00
1984-08-14T12:00+02:00[Europe/Stockholm]
1984-08-14T00:00Z

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:

Name	Description
A single map consisting of the following:	
<code>year</code>	An expression consisting of at least four digits that specifies the year.
<code>ordinalDay</code>	An integer between 1 and 366 that specifies the ordinal day of the year.
<code>hour</code>	An integer between 0 and 23 that specifies the hour of the day.
<code>minute</code>	An integer between 0 and 59 that specifies the number of minutes.

Name	Description
<code>second</code>	An integer between 0 and 59 that specifies the number of seconds.
<code>millisecond</code>	An integer between 0 and 999 that specifies the number of milliseconds.
<code>microsecond</code>	An integer between 0 and 999,999 that specifies the number of microseconds.
<code>nanosecond</code>	An integer between 0 and 999,999,999 that specifies the number of nanoseconds.
<code>timezone</code>	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.

## Example 161. datetime()

### Query

```
UNWIND [  
  datetime({year: 1984, ordinalDay: 202, hour: 12, minute: 31, second: 14, millisecond: 645}),  
  datetime({year: 1984, ordinalDay: 202, hour: 12, minute: 31, second: 14, timezone: '+01:00'}),  
  datetime({year: 1984, ordinalDay: 202, timezone: 'Europe/Stockholm'}),  
  datetime({year: 1984, ordinalDay: 202})  
] AS theDate  
RETURN theDate
```

### Table 405. Result

theDate
1984-07-20T12:31:14.645Z
1984-07-20T12:31:14+01:00
1984-07-20T00:00+02:00[Europe/Stockholm]
1984-07-20T00:00Z

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:

Name	Description
<code>temporalValue</code>	A string representing a temporal value.

### 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.

## Example 162. datetime()

### Query

```
UNWIND [  
  datetime('2015-07-21T21:40:32.142+0100'),  
  datetime('2015-W30-2T214032.142Z'),  
  datetime('2015T214032-0100'),  
  datetime('20150721T21:40-01:30'),  
  datetime('2015-W30T2140-02'),  
  datetime('2015202T21+18:00'),  
  datetime('2015-07-21T21:40:32.142[Europe/London]'),  
  datetime('2015-07-21T21:40:32.142-04[America/New_York]')  
] AS theDate  
RETURN theDate
```

### Table 406. Result

theDate
2015-07-21T21:40:32.142+01:00
2015-07-21T21:40:32.142Z
2015-01-01T21:40:32-01:00
2015-07-21T21:40-01:30
2015-07-20T21:40-02:00
2015-07-21T21:00+18:00
2015-07-21T21:40:32.142+01:00[Europe/London]
2015-07-21T21:40:32.142-04:00[America/New_York]
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:

Name	Description
A single map consisting of the following:	



Name	Description
<code>datetime</code>	A <code>DateTime</code> value.
<code>date</code>	A <code>Date</code> value.
<code>time</code>	A <code>Time</code> value.
<code>year</code>	An expression consisting of at least four digits that specifies the year.
<code>month</code>	An integer between 1 and 12 that specifies the month.
<code>day</code>	An integer between 1 and 31 that specifies the day of the month.
<code>week</code>	An integer between 1 and 53 that specifies the week.
<code>dayOfWeek</code>	An integer between 1 and 7 that specifies the day of the week.
<code>quarter</code>	An integer between 1 and 4 that specifies the quarter.
<code>dayOfQuarter</code>	An integer between 1 and 92 that specifies the day of the quarter.
<code>ordinalDay</code>	An integer between 1 and 366 that specifies the ordinal day of the year.
<code>hour</code>	An integer between 0 and 23 that specifies the hour of the day.
<code>minute</code>	An integer between 0 and 59 that specifies the number of minutes.
<code>second</code>	An integer between 0 and 59 that specifies the number of seconds.
<code>millisecond</code>	An integer between 0 and 999 that specifies the number of milliseconds.
<code>microsecond</code>	An integer between 0 and 999,999 that specifies the number of microseconds.
<code>nanosecond</code>	An integer between 0 and 999,999,999 that specifies the number of nanoseconds.
<code>timezone</code>	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.

### Example 163. `datetime()`

The following query shows the various usages of `datetime({date [, year, ..., timezone]})`.

#### Query

```
WITH date({year: 1984, month: 10, day: 11}) AS dd
RETURN
  datetime({date: dd, hour: 10, minute: 10, second: 10}) AS dateHHMMSS,
  datetime({date: dd, hour: 10, minute: 10, second: 10, timezone: '+05:00'}) AS dateHHMSSTimezone,
  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
  dateDDHHMSSTimezone
```

Table 407. Result

dateHHMMSS	dateHHMSSTimezone	dateDDHHMMSS	dateDDHHMSSTimezone
1984-10-11T10:10:10Z	1984-10-11T10:10:10+05:00	1984-10-28T10:10:10Z	1984-10-28T10:10:10-10:00[Pacific/Honolulu]
Rows: 1			

### Example 164. `datetime()`

The following query shows the various usages of `datetime({time [, year, ..., timezone]})`.

#### Query

```
WITH time({hour: 12, minute: 31, second: 14, microsecond: 645876, timezone: '+01:00'}) AS tt
RETURN
  datetime({year: 1984, month: 10, day: 11, time: tt}) AS YYYYMMDDTime,
  datetime({year: 1984, month: 10, day: 11, time: tt, timezone: '+05:00'}) AS YYYYMMDDTimeTimezone,
  datetime({year: 1984, month: 10, day: 11, time: tt, second: 42}) AS YYYYMMDDTimeSS,
  datetime({year: 1984, month: 10, day: 11, time: tt, second: 42, timezone: 'Pacific/Honolulu'}) AS
  YYYYMMDDTimeSSTimezone
```

Table 408. Result

YYYYMMDDTime	YYYYMMDDTimeTimezone	YYYYMMDDTimeSS	YYYYMMDDTimeSSTimezone
1984-10-11T12:31:14.645876+01:00	1984-10-11T16:31:14.645876+05:00	1984-10-11T12:31:42.645876+01:00	1984-10-11T01:31:42.645876-10:00[Pacific/Honolulu]
Rows: 1			

### Example 165. datetime()

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 409. Result

dateTime	dateTimeTimezone	dateTimeDDSS	dateTimeDDSSTimezone
1984-10-11T12:31:14.645Z	1984-10-11T12:31:14.645+05:00	1984-10-28T12:31:42.645Z	1984-10-28T12:31:42.645-10:00[Pacific/Honolulu]
Rows: 1			

### Example 166. datetime()

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 410. Result

dateTime	dateTimeTimezone	dateTimeDDSS	dateTimeDDSSTimezone
1984-10-11T12:00+01:00[Europe/Stockholm]	1984-10-11T16:00+05:00	1984-10-28T12:00:42+01:00[Europe/Stockholm]	1984-10-28T01:00:42-10:00[Pacific/Honolulu]
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:

Name	Description
A single map consisting of the following:	
<code>epochSeconds</code>	A numeric value representing the number of seconds from the UNIX epoch in the UTC time zone.
<code>epochMillis</code>	A numeric value representing the number of milliseconds from the UNIX epoch in the UTC time zone.

### Considerations:

`epochSeconds/epochMillis` may be used in conjunction with `nanosecond`.

### Example 167. `datetime()`

#### Query

```
RETURN datetime({epochSeconds: timestamp() / 1000, nanosecond: 23}) AS theDate
```

#### Table 411. Result

theDate
2022-06-14T10:02:30.000000023Z
Rows: 1

## Example 168. datetime()

### Query

```
RETURN datetime({epochMillis: 424797300000}) AS theDate
```

### Table 412. Result

theDate
1983-06-18T15:15Z
Rows: 1

## 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 string is `'year'` in order to ensure the returned value has the day set to `x` instead of the default day (which is `1`).

### Syntax:

```
datetime.truncate(unit [, temporalInstantValue [, mapOfComponents ] ])
```

### Returns:

A *DateTime*.

### Arguments:

Name	Description
<code>unit</code>	A string expression evaluating to one of the following strings: <code>'millennium'</code> , <code>'century'</code> , <code>'decade'</code> , <code>'year'</code> , <code>'weekYear'</code> , <code>'quarter'</code> , <code>'month'</code> , <code>'week'</code> , <code>'day'</code> , <code>'hour'</code> , <code>'minute'</code> , <code>'second'</code> , <code>'millisecond'</code> , <code>'microsecond'</code> .
<code>temporalInstantValue</code>	An expression of one of the following types: <i>DateTime</i> , <i>LocalDateTime</i> , <i>Date</i> .

Name	Description
<code>mapOfComponents</code>	An expression evaluating to a map containing components less significant than <code>unit</code> . During truncation, a time zone can be attached or overridden using the key <code>timezone</code> .

### Considerations:

`temporalInstantValue` cannot be a `Date` value if `unit` is one of: `'hour'`, `'minute'`, `'second'`, `'millisecond'`, `'microsecond'`.

The time zone of `temporalInstantValue` may be overridden; for example, `datetime.truncate('minute', input, {timezone: '+0200'})`.

If `temporalInstantValue` is one of `Time`, `DateTime` — a value with a time zone — and the time zone is overridden, no time conversion occurs.

If `temporalInstantValue` is one of `LocalDateTime`, `Date` — a value without a time zone — and the time zone is not overridden, the configured default time zone will be used.

Any component that is provided in `mapOfComponents` must be less significant than `unit`; i.e. if `unit` is `'day'`, `mapOfComponents` cannot contain information pertaining to a month.

Any component that is not contained in `mapOfComponents` and which is less significant than `unit` will be set to its `minimal value`.

If `mapOfComponents` is not provided, all components of the returned value which are less significant than `unit` will be set to their default values.

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())`.

### Example 169. `datetime()`

#### Query

```
WITH
  datetime({
    year:2017, month:11, day:11,
    hour:12, minute:31, second:14, nanosecond: 645876123,
    timezone: '+03:00'
  }) AS d
RETURN
  datetime.truncate('millennium', d, {timezone: 'Europe/Stockholm'}) AS truncMillenium,
  datetime.truncate('year', d, {day: 5}) AS truncYear,
  datetime.truncate('month', d) AS truncMonth,
  datetime.truncate('day', d, {millisecond: 2}) AS truncDay,
  datetime.truncate('hour', d) AS truncHour,
  datetime.truncate('second', d) AS truncSecond
```

Table 413. Result

truncMillenium	truncYear	truncMonth	truncDay	truncHour	truncSecond
2000-01-01T00:00+01:00[Europe/Stockholm]	2017-01-05T00:00+03:00	2017-11-01T00:00+03:00	2017-11-11T00:00:00.002+03:00	2017-11-11T12:00+03:00	2017-11-11T12:31:14+03:00
Rows: 1					

# localdatetime()

Details for using the `localdatetime()` function.

- [Getting the current \*LocalDateTime\*](#)
  - `localdatetime.transaction()`
  - `localdatetime.statement()`
  - `localdatetime.realtime()`
- [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:

Name	Description
A single map consisting of the following:	
<code>timezone</code>	A string expression that represents the <a href="#">time zone</a> .

### Considerations:

If no parameters are provided, `localdatetime()` must be invoked (`localdatetime({})` is invalid).

### Example 170. `localdatetime()`

#### Query

```
RETURN localdatetime() AS now
```

The current local date and time (i.e. in the local time zone) is returned.

#### Table 414. Result

now
2022-06-14T10:02:30.447
Rows: 1

### Example 171. `localdatetime()`

#### Query

```
RETURN localdatetime({timezone: 'America/Los Angeles'}) AS now
```

The current local date and time in California is returned.

#### Table 415. Result

now
2022-06-14T03:02:30.482
Rows: 1

### `localdatetime.transaction()`

`localdatetime.transaction()` returns the current `LocalDateTime` value using the `transaction` clock. This value will be the same for each invocation within the same transaction. However, a different value may be produced for different transactions.

#### Syntax:

```
localdatetime.transaction([{{timezone}}])
```

#### Returns:

A `LocalDateTime`.

#### Arguments:



Name	Description
<code>timezone</code>	A string expression that represents the <a href="#">time zone</a> .

### Example 172. `localdatetime.transaction()`

#### Query

```
RETURN localdatetime.transaction() AS now
```

#### Table 416. Result

now
<code>2022-06-14T10:02:30.532</code>
Rows: 1

### `localdatetime.statement()`

`localdatetime.statement()` returns the current `LocalDateTime` value using the `statement` clock. This value will be the same for each invocation within the same statement. However, a different value may be produced for different statements within the same transaction.

#### Syntax:

```
localdatetime.statement([[timezone]])
```

#### Returns:

A `LocalDateTime`.

#### Arguments:

Name	Description
<code>timezone</code>	A string expression that represents the <a href="#">time zone</a> .

### Example 173. `localdatetime.statement()`

#### Query

```
RETURN localdatetime.statement() AS now
```

#### Table 417. Result

now
2022-06-14T10:02:30.570
Rows: 1

### `localdatetime.realtime()`

`localdatetime.realtime()` returns the current `LocalDateTime` value using the `realtime` clock. This value will be the live clock of the system.

#### Syntax:

```
localdatetime.realtime([{{timezone}}])
```

#### Returns:

A `LocalDateTime`.

#### Arguments:

Name	Description
<code>timezone</code>	A string expression that represents the <code>time zone</code> .

### Example 174. `localdatetime.realtime()`

#### Query

```
RETURN localdatetime.realtime() AS now
```

#### Table 418. Result

now
2022-06-14T10:02:30.647817
Rows: 1

## Example 175. `localdatetime.realtime()`

### Query

```
RETURN localdatetime.realtime('America/Los Angeles') AS nowInLA
```

### Table 419. Result

nowInLA
2022-06-14T03:02:30.691099
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:

Name	Description
A single map consisting of the following:	
<code>year</code>	An expression consisting of at least four digits that specifies the year.
<code>month</code>	An integer between 1 and 12 that specifies the month.
<code>day</code>	An integer between 1 and 31 that specifies the day of the month.
<code>hour</code>	An integer between 0 and 23 that specifies the hour of the day.
<code>minute</code>	An integer between 0 and 59 that specifies the number of minutes.
<code>second</code>	An integer between 0 and 59 that specifies the number of seconds.
<code>millisecond</code>	An integer between 0 and 999 that specifies the number of milliseconds.
<code>microsecond</code>	An integer between 0 and 999,999 that specifies the number of microseconds.

Name	Description
<code>nanosecond</code>	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.

### Example 176. `localdatetime.realtime()`

#### Query

```
RETURN
  localdatetime({
    year: 1984, month: 10, day: 11,
    hour: 12, minute: 31, second: 14, millisecond: 123, microsecond: 456, nanosecond: 789
  }) AS theDate
```

#### Table 420. Result

theDate
1984-10-11T12:31:14.123456789
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:

Name	Description
A single map consisting of the following:	
<code>year</code>	An expression consisting of at least four digits that specifies the year.
<code>week</code>	An integer between 1 and 53 that specifies the week.
<code>dayOfWeek</code>	An integer between 1 and 7 that specifies the day of the week.
<code>hour</code>	An integer between 0 and 23 that specifies the hour of the day.
<code>minute</code>	An integer between 0 and 59 that specifies the number of minutes.
<code>second</code>	An integer between 0 and 59 that specifies the number of seconds.
<code>millisecond</code>	An integer between 0 and 999 that specifies the number of milliseconds.
<code>microsecond</code>	An integer between 0 and 999,999 that specifies the number of microseconds.
<code>nanosecond</code>	An integer between 0 and 999,999,999 that specifies the number of nanoseconds.

### Considerations:

The week component will default to 1 if `week` is omitted.

The day of the week component will default to 1 if `dayOfWeek` is omitted.

The hour component will default to 0 if `hour` is omitted.

The minute component will default to 0 if `minute` is omitted.

The second component will default to 0 if `second` is omitted.

Any missing `millisecond`, `microsecond` or `nanosecond` values will default to 0.

If `millisecond`, `microsecond` and `nanosecond` are given in combination (as part of the same set of parameters), the individual values must be in the range 0 to 999.

The least significant components in the set `year`, `week`, `dayOfWeek`, `hour`, `minute`, and `second` may be omitted; i.e. it is possible to specify only `year`, `week` and `dayOfWeek`, but specifying `year`, `week`, `dayOfWeek` and `minute` is not permitted.

One or more of `millisecond`, `microsecond` and `nanosecond` can only be specified as long as `second` is also specified.

## Example 177. localtime()

### Query

```
RETURN
  localtime({
    year: 1984, week: 10, dayOfWeek: 3,
    hour: 12, minute: 31, second: 14, millisecond: 645
  }) AS theDate
```

Table 421. Result

theDate
1984-03-07T12:31:14.645
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:

Name	Description
A single map consisting of the following:	
<code>year</code>	An expression consisting of at least four digits that specifies the year.
<code>quarter</code>	An integer between 1 and 4 that specifies the quarter.
<code>dayOfQuarter</code>	An integer between 1 and 92 that specifies the day of the quarter.
<code>hour</code>	An integer between 0 and 23 that specifies the hour of the day.
<code>minute</code>	An integer between 0 and 59 that specifies the number of minutes.
<code>second</code>	An integer between 0 and 59 that specifies the number of seconds.

Name	Description
<code>millisecond</code>	An integer between 0 and 999 that specifies the number of milliseconds.
<code>microsecond</code>	An integer between 0 and 999,999 that specifies the number of microseconds.
<code>nanosecond</code>	An integer between 0 and 999,999,999 that specifies the number of nanoseconds.

### Considerations:

The quarter component will default to 1 if `quarter` is omitted.

The day of the quarter component will default to 1 if `dayOfQuarter` is omitted.

The hour component will default to 0 if `hour` is omitted.

The minute component will default to 0 if `minute` is omitted.

The second component will default to 0 if `second` is omitted.

Any missing `millisecond`, `microsecond` or `nanosecond` values will default to 0.

If `millisecond`, `microsecond` and `nanosecond` are given in combination (as part of the same set of parameters), the individual values must be in the range 0 to 999.

The least significant components in the set `year`, `quarter`, `dayOfQuarter`, `hour`, `minute`, and `second` may be omitted; i.e. it is possible to specify only `year`, `quarter` and `dayOfQuarter`, but specifying `year`, `quarter`, `dayOfQuarter` and `minute` is not permitted.

One or more of `millisecond`, `microsecond` and `nanosecond` can only be specified as long as `second` is also specified.

### Example 178. `localdatetime()`

#### Query

```
RETURN
  localdatetime({
    year: 1984, quarter: 3, dayOfQuarter: 45,
    hour: 12, minute: 31, second: 14, nanosecond: 645876123
  }) AS theDate
```

#### Table 422. Result

theDate
1984-08-14T12:31:14.645876123
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:

Name	Description
A single map consisting of the following:	
<code>year</code>	An expression consisting of at least four digits that specifies the year.
<code>ordinalDay</code>	An integer between 1 and 366 that specifies the ordinal day of the year.
<code>hour</code>	An integer between 0 and 23 that specifies the hour of the day.
<code>minute</code>	An integer between 0 and 59 that specifies the number of minutes.
<code>second</code>	An integer between 0 and 59 that specifies the number of seconds.
<code>millisecond</code>	An integer between 0 and 999 that specifies the number of milliseconds.
<code>microsecond</code>	An integer between 0 and 999,999 that specifies the number of microseconds.
<code>nanosecond</code>	An integer between 0 and 999,999,999 that specifies the number of nanoseconds.

## Considerations:

The *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.



## Example 179. `localdatetime()`

### Query

```
RETURN
  localdatetime({
    year: 1984, ordinalDay: 202,
    hour: 12, minute: 31, second: 14, microsecond: 645876
  }) AS theDate
```

### Table 423. Result

theDate
1984-07-20T12:31:14.645876
Rows: 1

## 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:

Name	Description
<code>temporalValue</code>	A string representing a temporal value.

### 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.

## Example 180. localtime()

### Query

```
UNWIND [  
  localtime('2015-07-21T21:40:32.142'),  
  localtime('2015-W30-2T214032.142'),  
  localtime('2015-202T21:40:32'),  
  localtime('2015202T21')  
] AS theDate  
RETURN theDate
```

### Table 424. Result

theDate
2015-07-21T21:40:32.142
2015-07-21T21:40:32.142
2015-07-21T21:40:32
2015-07-21T21:00

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]}) | localtime({date [, year, ..., nanosecond]}) |  
localdatetime({time [, year, ..., nanosecond]}) | localtime({date, time [, year, ..., nanosecond]})
```

### Returns:

A `LocalDateTime`.

### Arguments:

Name	Description
A single map consisting of the following:	
<code>datetime</code>	A <code>DateTime</code> value.
<code>date</code>	A <code>Date</code> value.
<code>time</code>	A <code>Time</code> value.
<code>year</code>	An expression consisting of at least four digits that specifies the year.
<code>month</code>	An integer between 1 and 12 that specifies the month.

Name	Description
<code>day</code>	An integer between 1 and 31 that specifies the day of the month.
<code>week</code>	An integer between 1 and 53 that specifies the week.
<code>dayOfWeek</code>	An integer between 1 and 7 that specifies the day of the week.
<code>quarter</code>	An integer between 1 and 4 that specifies the quarter.
<code>dayOfQuarter</code>	An integer between 1 and 92 that specifies the day of the quarter.
<code>ordinalDay</code>	An integer between 1 and 366 that specifies the ordinal day of the year.
<code>hour</code>	An integer between 0 and 23 that specifies the hour of the day.
<code>minute</code>	An integer between 0 and 59 that specifies the number of minutes.
<code>second</code>	An integer between 0 and 59 that specifies the number of seconds.
<code>millisecond</code>	An integer between 0 and 999 that specifies the number of milliseconds.
<code>microsecond</code>	An integer between 0 and 999,999 that specifies the number of microseconds.
<code>nanosecond</code>	An integer between 0 and 999,999,999 that specifies the number of nanoseconds.

### Considerations:

If any of the optional parameters are provided, these will override the corresponding components of `datetime`, `date` and/or `time`.

`localdatetime(dd)` may be written instead of `localdatetime({datetime: dd})`.

### Example 181. localdatetime()

The following query shows the various usages of `localdatetime({date [, year, ..., nanosecond]})`.

#### Query

```
WITH date({year: 1984, month: 10, day: 11}) AS dd
RETURN
  localdatetime({date: dd, hour: 10, minute: 10, second: 10}) AS dateHHMMSS,
  localdatetime({date: dd, day: 28, hour: 10, minute: 10, second: 10}) AS dateDDHHMMSS
```

#### Table 425. Result

dateHHMMSS	dateDDHHMMSS
1984-10-11T10:10:10	1984-10-28T10:10:10
Rows: 1	

### Example 182. localdatetime()

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 426. Result

YYYYMMDDTime	YYYYMMDDTimeSS
1984-10-11T12:31:14.645876	1984-10-11T12:31:42.645876
Rows: 1	

### Example 183. `localdatetime()`

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 427. Result

dateTime	dateTimeDDSS
1984-10-11T12:31:14.645876	1984-10-28T12:31:42.645876
Rows: 1	

### Example 184. `localdatetime()`

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 428. Result

dateTime	dateTimeDDSS
1984-10-11T12:00	1984-10-28T12:00:42
Rows: 1	

## 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 string is `'year'` in order to ensure the returned value has the day set to `x` instead of the default day (which is `1`).

#### Syntax:

```
localdatetime.truncate(unit [, temporalInstantValue [, mapOfComponents ] ])
```

#### Returns:

A `LocalDateTime`.

#### Arguments:

Name	Description
<code>unit</code>	A string expression evaluating to one of the following strings: <code>'millennium'</code> , <code>'century'</code> , <code>'decade'</code> , <code>'year'</code> , <code>'weekYear'</code> , <code>'quarter'</code> , <code>'month'</code> , <code>'week'</code> , <code>'day'</code> , <code>'hour'</code> , <code>'minute'</code> , <code>'second'</code> , <code>'millisecond'</code> , <code>'microsecond'</code> .
<code>temporalInstantValue</code>	An expression of one of the following types: <code>DateTime</code> , <code>LocalDateTime</code> , <code>Date</code> .
<code>mapOfComponents</code>	An expression evaluating to a map containing components less significant than <code>unit</code> .

#### Considerations:

`temporalInstantValue` cannot be a `Date` value if `unit` is one of: `'hour'`, `'minute'`, `'second'`, `'millisecond'`, `'microsecond'`.

Any component that is provided in `mapOfComponents` must be less significant than `unit`; i.e. if `unit` is `'day'`, `mapOfComponents` cannot contain information pertaining to a month.

Any component that is not contained in `mapOfComponents` and which is less significant than `unit` will be set to its [minimal value](#).

If `mapOfComponents` is not provided, all components of the returned value which are less significant than `unit` will be set to their default values.

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())`.

## Example 185. `localdatetime.truncate()`

### Query

```
WITH
  localdatetime({
    year: 2017, month: 11, day: 11,
    hour: 12, minute: 31, second: 14, nanosecond: 645876123
  }) AS d
RETURN
  localdatetime.truncate('millennium', d) AS truncMillenium,
  localdatetime.truncate('year', d, {day: 2}) AS truncYear,
  localdatetime.truncate('month', d) AS truncMonth,
  localdatetime.truncate('day', d) AS truncDay,
  localdatetime.truncate('hour', d, {nanosecond: 2}) AS truncHour,
  localdatetime.truncate('second', d) AS truncSecond
```

Table 429. Result

truncMillenium	truncYear	truncMonth	truncDay	truncHour	truncSecond
2000-01-01T00:00	2017-01-02T00:00	2017-11-01T00:00	2017-11-11T00:00	2017-11-11T12:00:00.00000002	2017-11-11T12:31:14

Rows: 1

## `localtime()`

Details for using the `localtime()` function.

- [Getting the current `LocalTime`](#)
  - `localtime.transaction()`
  - `localtime.statement()`
  - `localtime.realtime()`
- [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:

Name	Description
A single map consisting of the following:	
<code>timezone</code>	A string expression that represents the <a href="#">time zone</a> .

### Considerations:

If no parameters are provided, `localtime()` must be invoked (`localtime({})` is invalid).

### Example 186. `localtime()`

#### Query

```
RETURN localtime() AS now
```

The current local time (i.e. in the local time zone) is returned.

#### Table 430. Result

now
10:02:31.596
Rows: 1

### Example 187. `localtime()`

#### Query

```
RETURN localtime({timezone: 'America/Los Angeles'}) AS nowInLA
```

The current local time in California is returned.

#### Table 431. Result

nowInLA
03:02:31.629
Rows: 1

### `localtime.transaction()`

`localtime.transaction()` returns the current `LocalTime` value using the `transaction` clock. This value will be the same for each invocation within the same transaction. However, a different value may be produced for different transactions.



### Syntax:

```
localtime.transaction([[timezone]])
```

### Returns:

A LocalTime.

### Arguments:

Name	Description
<code>timezone</code>	A string expression that represents the <a href="#">time zone</a> .

### Example 188. localtime.transaction()

#### Query

```
RETURN localtime.transaction() AS now
```

#### Table 432. Result

now

10:02:31.662

Rows: 1

### localtime.statement()

`localtime.statement()` returns the current *LocalTime* value using the `statement` clock. This value will be the same for each invocation within the same statement. However, a different value may be produced for different statements within the same transaction.

### Syntax:

```
localtime.statement([[timezone]])
```

### Returns:

A LocalTime.

### Arguments:

Name	Description
<code>timezone</code>	A string expression that represents the <a href="#">time zone</a> .

### Example 189. localtime.statement()

#### Query

```
RETURN localtime.statement() AS now
```

#### Table 433. Result

now
10:02:31.697
Rows: 1

### Example 190. localtime.statement()

#### Query

```
RETURN localtime.statement('America/Los Angeles') AS nowInLA
```

#### Table 434. Result

nowInLA
03:02:31.737
Rows: 1

### localtime.realtime()

`localtime.realtime()` returns the current `LocalTime` value using the `realtime` clock. This value will be the live clock of the system.

#### Syntax:

```
localtime.realtime([{timezone}])
```

#### Returns:

A `LocalTime`.

#### Arguments:

Name	Description
<code>timezone</code>	A string expression that represents the <code>time zone</code> .

## Example 191. localtime.realtime()

### Query

```
RETURN localtime.realtime() AS now
```

### Table 435. Result

now
10:02:31.806895
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:

Name	Description
A single map consisting of the following:	
<code>hour</code>	An integer between 0 and 23 that specifies the hour of the day.
<code>minute</code>	An integer between 0 and 59 that specifies the number of minutes.
<code>second</code>	An integer between 0 and 59 that specifies the number of seconds.
<code>millisecond</code>	An integer between 0 and 999 that specifies the number of milliseconds.
<code>microsecond</code>	An integer between 0 and 999,999 that specifies the number of microseconds.
<code>nanosecond</code>	An integer between 0 and 999,999,999 that specifies the number of nanoseconds.

### Considerations:

The `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.

### Example 192. `localtime()`

#### Query

```
UNWIND [  
  localtime({hour: 12, minute: 31, second: 14, nanosecond: 789, millisecond: 123, microsecond: 456}),  
  localtime({hour: 12, minute: 31, second: 14}),  
  localtime({hour: 12})  
] AS theTime  
RETURN theTime
```

Table 436. Result

theTime
12:31:14.123456789
12:31:14
12:00

Rows: 3

## Creating a `LocalTime` from a string

`localtime()` returns the `LocalTime` value obtained by parsing a string representation of a temporal value.

#### Syntax:

```
localtime(temporalValue)
```

#### Returns:

A `LocalTime`.

#### Arguments:

Name	Description
<code>temporalValue</code>	A string representing a temporal value.

#### Considerations:

`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.

### Example 193. `localtime()`

#### Query

```
UNWIND [  
  localtime('21:40:32.142'),  
  localtime('214032.142'),  
  localtime('21:40'),  
  localtime('21')  
] AS theTime  
RETURN theTime
```

#### Table 437. Result

theTime
21:40:32.142
21:40:32.142
21:40
21:00
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:

Name	Description
A single map consisting of the following:	
<code>time</code>	A <code>Time</code> value.
<code>hour</code>	An integer between 0 and 23 that specifies the hour of the day.

Name	Description
<code>minute</code>	An integer between 0 and 59 that specifies the number of minutes.
<code>second</code>	An integer between 0 and 59 that specifies the number of seconds.
<code>millisecond</code>	An integer between 0 and 999 that specifies the number of milliseconds.
<code>microsecond</code>	An integer between 0 and 999,999 that specifies the number of microseconds.
<code>nanosecond</code>	An integer between 0 and 999,999,999 that specifies the number of nanoseconds.

### Considerations:

If any of the optional parameters are provided, these will override the corresponding components of `time`.

`localtime(tt)` may be written instead of `localtime({time: tt})`.

### Example 194. `localtime()`

#### Query

```
WITH time({hour: 12, minute: 31, second: 14, microsecond: 645876, timezone: '+01:00'}) AS tt
RETURN
  localtime({time: tt}) AS timeOnly,
  localtime({time: tt, second: 42}) AS timeSS
```

Table 438. Result

timeOnly	timeSS
12:31:14.645876	12:31:42.645876
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 string is `'hour'` in order to ensure the returned value has the `minute` set to `x` instead of the default `minute` (which is 1).

### Syntax:

```
localtime.truncate(unit [, temporalInstantValue [, mapOfComponents ] ])
```

### Returns:

A LocalTime.

### Arguments:

Name	Description
<code>unit</code>	A string expression evaluating to one of the following strings: 'day', 'hour', 'minute', 'second', 'millisecond', 'microsecond'.
<code>temporalInstantValue</code>	An expression of one of the following types: <code>DateTime</code> , <code>LocalDateTime</code> , <code>Time</code> , <code>LocalTime</code> .
<code>mapOfComponents</code>	An expression evaluating to a map containing components less significant than <code>unit</code> .

### Considerations:

Truncating time to day — i.e. `unit` is 'day' — is supported, and yields midnight at the start of the day (00:00), regardless of the value of `temporalInstantValue`. However, the time zone of `temporalInstantValue` is retained.

Any component that is provided in `mapOfComponents` must be less significant than `unit`; i.e. if `unit` is 'second', `mapOfComponents` cannot contain information pertaining to a minute.

Any component that is not contained in `mapOfComponents` and which is less significant than `unit` will be set to its [minimal value](#).

If `mapOfComponents` is not provided, all components of the returned value which are less significant than `unit` will be set to their default values.

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())`.

## Example 195. localtime.truncate()

### 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 439. Result

truncDay	truncHour	truncMinute	truncSecond	truncMillisecond	truncMicrosecond
00:00	12:00	12:31:00.002	12:31:14	12:31:14.645	12:31:14.645876

Rows: 1

## time()

Details for using the `time()` function.

- [Getting the current Time](#)
  - [time.transaction\(\)](#)
  - [time.statement\(\)](#)
  - [time.realtime\(\)](#)
- [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([[timezone]])
```

Returns:

A Time.

Arguments:



Name	Description
A single map consisting of the following:	
<code>timezone</code>	A string expression that represents the <a href="#">time zone</a> .

### Considerations:

If no parameters are provided, `time()` must be invoked (`time({})` is invalid).

### Example 196. `time()`

#### Query

```
RETURN time() AS currentTime
```

The current time of day using the local time zone is returned.

#### Table 440. Result

currentTime
10:02:32.192Z
Rows: 1

### Example 197. `time()`

#### Query

```
RETURN time({timezone: 'America/Los Angeles'}) AS currentTimeInLA
```

The current time of day in California is returned.

#### Table 441. Result

currentTimeInLA
03:02:32.233-07:00
Rows: 1

### `time.transaction()`

`time.transaction()` returns the current *Time* value using the `transaction` clock. This value will be the same for each invocation within the same transaction. However, a different value may be produced for different transactions.

### Syntax:

```
time.transaction([[timezone]])
```

### Returns:

A Time.

### Arguments:

Name	Description
<code>timezone</code>	A string expression that represents the <a href="#">time zone</a> .

### Example 198. `time.transaction()`

#### Query

```
RETURN time.transaction() AS currentTime
```

#### Table 442. Result

currentTime
<code>10:02:32.276Z</code>

Rows: 1

### `time.statement()`

`time.statement()` returns the current *Time* value using the `statement` clock. This value will be the same for each invocation within the same statement. However, a different value may be produced for different statements within the same transaction.

### Syntax:

```
time.statement([{{timezone}}])
```

### Returns:

A Time.

### Arguments:

Name	Description
<code>timezone</code>	A string expression that represents the <a href="#">time zone</a> .

### Example 199. time.statement()

#### Query

```
RETURN time.statement() AS currentTime
```

#### Table 443. Result

currentTime
10:02:32.317Z
Rows: 1

### Example 200. time.statement()

#### Query

```
RETURN time.statement('America/Los Angeles') AS currentTimeInLA
```

#### Table 444. Result

currentTimeInLA
03:02:32.351-07:00
Rows: 1

## time.realtime()

`time.realtime()` returns the current *Time* value using the *realtime* clock. This value will be the live clock of the system.

### Syntax:

```
time.realtime([[timezone]])
```

### Returns:

A *Time*.

### Arguments:

Name	Description
<code>timezone</code>	A string expression that represents the <i>time zone</i> .

## Example 201. time.realtime()

### Query

```
RETURN time.realtime() AS currentTime
```

### Table 445. Result

currentTime
10:02:32.436948Z
Rows: 1

## Creating a Time

`time()` returns a *Time* value with the specified *hour*, *minute*, *second*, *millisecond*, *microsecond*, *nanosecond* and *timezone* component values.

### Syntax:

```
time({hour [, minute, second, millisecond, microsecond, nanosecond, timezone]})
```

### Returns:

A *Time*.

### Arguments:

Name	Description
A single map consisting of the following:	
<code>hour</code>	An integer between 0 and 23 that specifies the hour of the day.
<code>minute</code>	An integer between 0 and 59 that specifies the number of minutes.
<code>second</code>	An integer between 0 and 59 that specifies the number of seconds.
<code>millisecond</code>	An integer between 0 and 999 that specifies the number of milliseconds.
<code>microsecond</code>	An integer between 0 and 999,999 that specifies the number of microseconds.
<code>nanosecond</code>	An integer between 0 and 999,999,999 that specifies the number of nanoseconds.
<code>timezone</code>	An expression that specifies the time zone.

### Considerations:

The `hour` component will default to `0` if `hour` is omitted.

The `minute` component will default to `0` if `minute` is omitted.

The `second` component will default to `0` if `second` is omitted.

Any missing `millisecond`, `microsecond` or `nanosecond` values will default to `0`.

The `timezone` component will default to the configured default time zone if `timezone` is omitted.

If `millisecond`, `microsecond` and `nanosecond` are given in combination (as part of the same set of parameters), the individual values must be in the range `0` to `999`.

The least significant components in the set `hour`, `minute`, and `second` may be omitted; i.e. it is possible to specify only `hour` and `minute`, but specifying `hour` and `second` is not permitted.

One or more of `millisecond`, `microsecond` and `nanosecond` can only be specified as long as `second` is also specified.

### Example 202. `time()`

#### Query

```
UNWIND [  
  time({hour: 12, minute: 31, second: 14, millisecond: 123, microsecond: 456, nanosecond: 789}),  
  time({hour: 12, minute: 31, second: 14, nanosecond: 645876123}),  
  time({hour: 12, minute: 31, second: 14, microsecond: 645876, timezone: '+01:00'}),  
  time({hour: 12, minute: 31, timezone: '+01:00'}),  
  time({hour: 12, timezone: '+01:00'})  
] AS theTime  
RETURN theTime
```

#### Table 446. Result

theTime
12:31:14.123456789Z
12:31:14.645876123Z
12:31:14.645876+01:00
12:31+01:00
12:00+01:00

Rows: 5

## Creating a *Time* from a string

`time()` returns the *Time* value obtained by parsing a string representation of a temporal value.

#### Syntax:

```
time(temporalValue)
```

#### Returns:

A *Time*.

## Arguments:

Name	Description
<code>temporalValue</code>	A string representing a temporal value.

## Considerations:

`temporalValue` must comply with the format defined for [times](#) and [time zones](#).

The `timezone` component will default to the configured default time zone if it is omitted.

`temporalValue` must denote a valid time; i.e. a `temporalValue` denoting `15:67` is invalid.

`time(null)` returns `null`.

## Example 203. `time()`

### Query

```
UNWIND [  
  time('21:40:32.142+0100'),  
  time('214032.142Z'),  
  time('21:40:32+01:00'),  
  time('214032-0100'),  
  time('21:40-01:30'),  
  time('2140-00:00'),  
  time('2140-02'),  
  time('22+18:00')  
] AS theTime  
RETURN theTime
```

### Table 447. Result

theTime
<code>21:40:32.142+01:00</code>
<code>21:40:32.142Z</code>
<code>21:40:32+01:00</code>
<code>21:40:32-01:00</code>
<code>21:40-01:30</code>
<code>21:40Z</code>
<code>21:40-02:00</code>
<code>22:00+18:00</code>
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:

Name	Description
A single map consisting of the following:	
<code>time</code>	A <i>Time</i> value.
<code>hour</code>	An integer between 0 and 23 that specifies the hour of the day.
<code>minute</code>	An integer between 0 and 59 that specifies the number of minutes.
<code>second</code>	An integer between 0 and 59 that specifies the number of seconds.
<code>millisecond</code>	An integer between 0 and 999 that specifies the number of milliseconds.
<code>microsecond</code>	An integer between 0 and 999,999 that specifies the number of microseconds.
<code>nanosecond</code>	An integer between 0 and 999,999,999 that specifies the number of nanoseconds.
<code>timezone</code>	An expression that specifies the time zone.

## Considerations:

If any of the optional parameters are provided, these will override the corresponding components of `time`.

`time(tt)` may be written instead of `time({time: tt})`.

Selecting a *Time* or *DateTime* value as the `time` component also selects its time zone. If a *LocalTime* or *LocalDateTime* is selected instead, the default time zone is used. In any case, the time zone can be overridden explicitly.

Selecting a *DateTime* or *Time* as the `time` component and overwriting the time zone will adjust the local time to keep the same point in time.

## Example 204. time()

### Query

```
WITH localtime({hour: 12, minute: 31, second: 14, microsecond: 645876}) AS tt
RETURN
  time({time: tt}) AS timeOnly,
  time({time: tt, timezone: '+05:00'}) AS timeTimezone,
  time({time: tt, second: 42}) AS timeSS,
  time({time: tt, second: 42, timezone: '+05:00'}) AS timeSSTimezone
```

Table 448. Result

timeOnly	timeTimezone	timeSS	timeSSTimezone
12:31:14.645876Z	12:31:14.645876+05:00	12:31:42.645876Z	12:31:42.645876+05:00
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 string is `'hour'` in order to ensure the returned value has the `minute` set to `x` instead of the default `minute` (which is 1).

### Syntax:

```
time.truncate(unit [, temporalInstantValue [, mapOfComponents ] ])
```

### Returns:

A *Time*.

### Arguments:

Name	Description
<code>unit</code>	A string expression evaluating to one of the following strings: <code>'day'</code> , <code>'hour'</code> , <code>'minute'</code> , <code>'second'</code> , <code>'millisecond'</code> , <code>'microsecond'</code> .
<code>temporalInstantValue</code>	An expression of one of the following types: <code>DateTime</code> , <code>LocalDateTime</code> , <code>Time</code> , <code>LocalTime</code> .



Name	Description
<code>mapOfComponents</code>	An expression evaluating to a map containing components less significant than <code>unit</code> . During truncation, a time zone can be attached or overridden using the key <code>timezone</code> .

### Considerations:

Truncating time to day — i.e. `unit` is `'day'` — is supported, and yields midnight at the start of the day (`00:00`), regardless of the value of `temporalInstantValue`. However, the time zone of `temporalInstantValue` is retained.

The time zone of `temporalInstantValue` may be overridden; for example, `time.truncate('minute', input, {timezone: '+0200'})`.

If `temporalInstantValue` is one of `Time`, `DateTime` — a value with a time zone — and the time zone is overridden, no time conversion occurs.

If `temporalInstantValue` is one of `LocalTime`, `LocalDateTime`, `Date` — a value without a time zone — and the time zone is not overridden, the configured default time zone will be used.

Any component that is provided in `mapOfComponents` must be less significant than `unit`; i.e. if `unit` is `'second'`, `mapOfComponents` cannot contain information pertaining to a `minute`.

Any component that is not contained in `mapOfComponents` and which is less significant than `unit` will be set to its `minimal value`.

If `mapOfComponents` is not provided, all components of the returned value which are less significant than `unit` will be set to their default values.

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())`.

### Example 205. time()

#### Query

```
WITH time({hour: 12, minute: 31, second: 14, nanosecond: 645876123, timezone: '-01:00'}) AS t
RETURN
  time.truncate('day', t) AS truncDay,
  time.truncate('hour', t) AS truncHour,
  time.truncate('minute', t) AS truncMinute,
  time.truncate('second', t) AS truncSecond,
  time.truncate('millisecond', t, {nanosecond: 2}) AS truncMillisecond,
  time.truncate('microsecond', t) AS truncMicrosecond
```

Table 449. Result

truncDay	truncHour	truncMinute	truncSecond	truncMillisecond	truncMicrosecond
<code>00:00-01:00</code>	<code>12:00-01:00</code>	<code>12:31-01:00</code>	<code>12:31:14-01:00</code>	<code>12:31:14.6450000-02-01:00</code>	<code>12:31:14.645876-01:00</code>

Rows: 1

# 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](#).

## Creating a *Duration* from duration components

`duration()` can construct a *Duration* from a map of its components in the same way as the temporal instant types.

- `years`
- `quarters`
- `months`
- `weeks`
- `days`
- `hours`
- `minutes`
- `seconds`
- `milliseconds`
- `microseconds`
- `nanoseconds`

**Syntax:**

```
duration([ {years, quarters, months, weeks, days, hours, minutes, seconds, milliseconds, microseconds, nanoseconds} ])
```

**Returns:**

A *Duration*.

**Arguments:**

Name	Description
A single map consisting of the following:	
years	A numeric expression.
quarters	A numeric expression.
months	A numeric expression.
weeks	A numeric expression.
days	A numeric expression.
hours	A numeric expression.
minutes	A numeric expression.
seconds	A numeric expression.
milliseconds	A numeric expression.
microseconds	A numeric expression.
nanoseconds	A numeric expression.

### Considerations:

At least one parameter must be provided (`duration()` and `duration({})` are invalid).

There is no constraint on how many of the parameters are provided.

It is possible to have a *Duration* where the amount of a smaller unit (e.g. `seconds`) exceeds the threshold of a larger unit (e.g. `days`).

The values of the parameters may be expressed as decimal fractions.

The values of the parameters may be arbitrarily large.

The values of the parameters may be negative.

## Example 206. duration()

### Query

```
UNWIND [  
  duration({days: 14, hours:16, minutes: 12}),  
  duration({months: 5, days: 1.5}),  
  duration({months: 0.75}),  
  duration({weeks: 2.5}),  
  duration({minutes: 1.5, seconds: 1, milliseconds: 123, microseconds: 456, nanoseconds: 789}),  
  duration({minutes: 1.5, seconds: 1, nanoseconds: 123456789})  
] AS aDuration  
RETURN aDuration
```

### Table 450. Result

aDuration
P14DT16H12M
P5M1DT12H
P22DT19H51M49.5S
P17DT12H
PT1M31.123456789S
PT1M31.123456789S

Rows: 6

## Creating a Duration from a string

`duration()` returns the *Duration* value obtained by parsing a string representation of a temporal amount.

### Syntax:

```
duration(temporalAmount)
```

### Returns:

A Duration.

### Arguments:

Name	Description
<code>temporalAmount</code>	A string representing a temporal amount.

### Considerations:

`temporalAmount` must comply with either the [unit based form](#) or [date-and-time based form](#) defined for Durations.

## Example 207. duration()

### Query

```
UNWIND [  
  duration("P14DT16H12M"),  
  duration("P5M1.5D"),  
  duration("P0.75M"),  
  duration("PT0.75M"),  
  duration("P2012-02-02T14:37:21.545")  
] AS aDuration  
RETURN aDuration
```

### Table 451. Result

aDuration
P14DT16H12M
P5M1DT12H
P22DT19H51M49.5S
PT45S
P2012Y2M2DT14H37M21.545S
Rows: 5

## Computing the *Duration* between two temporal instants

`duration()` has sub-functions which compute the *logical difference* (in days, months, etc) between two temporal instant values:

- `duration.between(a, b)`: Computes the difference in multiple components between instant `a` and instant `b`. This captures month, days, seconds and sub-seconds differences separately.
- `duration.inMonths(a, b)`: Computes the difference in whole months (or quarters or years) between instant `a` and instant `b`. This captures the difference as the total number of months. Any difference smaller than a whole month is disregarded.
- `duration.inDays(a, b)`: Computes the difference in whole days (or weeks) between instant `a` and instant `b`. This captures the difference as the total number of days. Any difference smaller than a whole day is disregarded.
- `duration.inSeconds(a, b)`: Computes the difference in seconds (and fractions of seconds, or minutes or hours) between instant `a` and instant `b`. This captures the difference as the total number of seconds.

### `duration.between()`

`duration.between()` returns the *Duration* value equal to the difference between the two given instants.

### Syntax:

```
duration.between(instant1, instant2)
```

## Returns:

A Duration.

## Arguments:

Name	Description
<code>instant1</code>	An expression returning any temporal instant type ( <i>Date</i> etc) that represents the starting instant.
<code>instant2</code>	An expression returning any temporal instant type ( <i>Date</i> etc) that represents the ending instant.

## Considerations:

If `instant2` occurs earlier than `instant1`, the resulting *Duration* will be negative.

If `instant1` has a time component and `instant2` does not, the time component of `instant2` is assumed to be midnight, and vice versa.

If `instant1` has a time zone component and `instant2` does not, the time zone component of `instant2` is assumed to be the same as that of `instant1`, and vice versa.

If `instant1` has a date component and `instant2` does not, the date component of `instant2` is assumed to be the same as that of `instant1`, and vice versa.

## Example 208. duration.between()

### Query

```
UNWIND [  
  duration.between(date("1984-10-11"), date("1985-11-25")),  
  duration.between(date("1985-11-25"), date("1984-10-11")),  
  duration.between(date("1984-10-11"), datetime("1984-10-12T21:40:32.142+0100")),  
  duration.between(date("2015-06-24"), localtime("14:30")),  
  duration.between(localtime("14:30"), time("16:30+0100")),  
  duration.between(localdatetime("2015-07-21T21:40:32.142"), localdatetime("2016-07-21T21:45:22.142")),  
  duration.between(datetime({year: 2017, month: 10, day: 29, hour: 0, timezone: 'Europe/Stockholm'}),  
    datetime({year: 2017, month: 10, day: 29, hour: 0, timezone: 'Europe/London'}))  
] AS aDuration  
RETURN aDuration
```

### Table 452. Result

aDuration
P1Y1M14D
P-1Y-1M-14D
P1DT21H40M32.142S
PT14H30M
PT2H
P1YT4M50S
PT1H
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(instant1, instant2)
```

### Returns:

A *Duration*.

### Arguments:

Name	Description
<code>instant1</code>	An expression returning any temporal instant type ( <i>Date</i> etc) that represents the starting instant.
<code>instant2</code>	An expression returning any temporal instant type ( <i>Date</i> etc) that represents the ending instant.

## Considerations:

If `instant2` occurs earlier than `instant1`, the resulting `Duration` will be negative.

If `instant1` has a time component and `instant2` does not, the time component of `instant2` is assumed to be midnight, and vice versa.

If `instant1` has a time zone component and `instant2` does not, the time zone component of `instant2` is assumed to be the same as that of `instant1`, and vice versa.

If `instant1` has a date component and `instant2` does not, the date component of `instant2` is assumed to be the same as that of `instant1`, and vice versa.

Any difference smaller than a whole month is disregarded.

## Example 209. `duration.inMonths()`

### Query

```
UNWIND [  
  duration.inMonths(date("1984-10-11"), date("1985-11-25")),  
  duration.inMonths(date("1985-11-25"), date("1984-10-11")),  
  duration.inMonths(date("1984-10-11"), datetime("1984-10-12T21:40:32.142+0100")),  
  duration.inMonths(date("2015-06-24"), localtime("14:30")),  
  duration.inMonths(localdatetime("2015-07-21T21:40:32.142"), localdatetime("2016-07-  
21T21:45:22.142")),  
  duration.inMonths(datetime({year: 2017, month: 10, day: 29, hour: 0, timezone: 'Europe/Stockholm'}),  
  datetime({year: 2017, month: 10, day: 29, hour: 0, timezone: 'Europe/London'}))  
] AS aDuration  
RETURN aDuration
```

### Table 453. Result

aDuration
P1Y1M
P-1Y-1M
PT0S
PT0S
P1Y
PT0S
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:

Name	Description
<code>instant1</code>	An expression returning any temporal instant type (Date etc) that represents the starting instant.
<code>instant2</code>	An expression returning any temporal instant type (Date etc) that represents the ending instant.

### Considerations:

If `instant2` occurs earlier than `instant1`, the resulting Duration will be negative.

If `instant1` has a time component and `instant2` does not, the time component of `instant2` is assumed to be midnight, and vice versa.

If `instant1` has a time zone component and `instant2` does not, the time zone component of `instant2` is assumed to be the same as that of `instant1`, and vice versa.

If `instant1` has a date component and `instant2` does not, the date component of `instant2` is assumed to be the same as that of `instant1`, and vice versa.

Any difference smaller than a whole day is disregarded.

### Example 210. `duration.inDays()`

#### Query

```
UNWIND [  
  duration.inDays(date("1984-10-11"), date("1985-11-25")),  
  duration.inDays(date("1985-11-25"), date("1984-10-11")),  
  duration.inDays(date("1984-10-11"), datetime("1984-10-12T21:40:32.142+0100")),  
  duration.inDays(date("2015-06-24"), localtime("14:30")),  
  duration.inDays(localdatetime("2015-07-21T21:40:32.142"), localdatetime("2016-07-21T21:45:22.142")),  
  duration.inDays(datetime({year: 2017, month: 10, day: 29, hour: 0, timezone: 'Europe/Stockholm'}),  
    datetime({year: 2017, month: 10, day: 29, hour: 0, timezone: 'Europe/London'}))  
] AS aDuration  
RETURN aDuration
```

#### Table 454. Result

aDuration
P410D
P-410D
P1D
PT0S
P366D
PT0S
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:

Name	Description
<code>instant1</code>	An expression returning any temporal instant type ( <i>Date</i> etc) that represents the starting instant.
<code>instant2</code>	An expression returning any temporal instant type ( <i>Date</i> etc) that represents the ending instant.

### Considerations:

If `instant2` occurs earlier than `instant1`, the resulting *Duration* will be negative.

If `instant1` has a time component and `instant2` does not, the time component of `instant2` is assumed to be midnight, and vice versa.

If `instant1` has a time zone component and `instant2` does not, the time zone component of `instant2` is assumed to be the same as that of `instant1`, and vice versa.

If `instant1` has a date component and `instant2` does not, the date component of `instant2` is assumed to be the same as that of `instant1`, and vice versa.

## Example 211. duration.inSeconds()

### Query

```
UNWIND [  
  duration.inSeconds(date("1984-10-11"), date("1984-10-12")),  
  duration.inSeconds(date("1984-10-12"), date("1984-10-11")),  
  duration.inSeconds(date("1984-10-11"), datetime("1984-10-12T01:00:32.142+0100")),  
  duration.inSeconds(datetime("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 455. Result

aDuration
PT24H
PT-24H
PT25H32.142S
PT14H30M
PT1H
Rows: 5

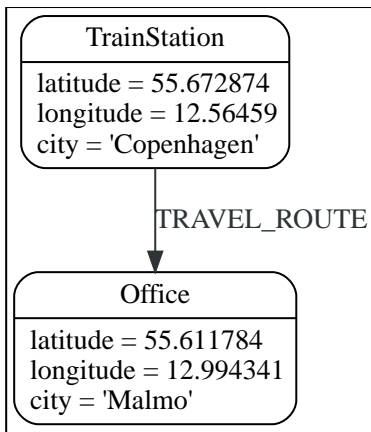
## Spatial functions

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.



## 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:

Name	Description
<code>point1</code>	A point in either a geographic or cartesian coordinate system.
<code>point2</code>	A point in the same CRS as <code>point1</code> .

## Considerations:

`point.distance(null, null)` return `null`.

`point.distance(null, point2)` return `null`.

`point.distance(point1, null)` return `null`.

Attempting to use points with different Coordinate Reference Systems (such as WGS 84 2D and WGS 84 3D) will return `null`.

## Example 212. `point.distance()`

### Query

```
WITH
  point({x: 2.3, y: 4.5, crs: 'cartesian'}) AS p1,
  point({x: 1.1, y: 5.4, crs: 'cartesian'}) AS p2
RETURN point.distance(p1,p2) AS dist
```

The distance between two 2D points in the Cartesian CRS is returned.

### Table 456. Result

dist
1.5

Rows: 1

## Example 213. `point.distance()`

### Query

```
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 457. Result

dist
1269.9148706779097

Rows: 1

### Example 214. point.distance()

#### Query

```
MATCH (t:TrainStation)-[:TRAVEL_ROUTE]->(o:Office)
WITH
  point({longitude: t.longitude, latitude: t.latitude}) AS trainPoint,
  point({longitude: o.longitude, latitude: o.latitude}) AS officePoint
RETURN round(point.distance(trainPoint, officePoint)) AS travelDistance
```

The distance between the train station in Copenhagen and the Neo4j office in Malmo is returned.

Table 458. Result

travelDistance
27842.0
Rows: 1

### Example 215. point.distance()

#### Query

```
RETURN point.distance(null, point({longitude: 56.7, latitude: 12.78})) AS d
```

If `null` is provided as one or both of the arguments, `null` is returned.

Table 459. Result

d
<null>
Rows: 1

## 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:

Name	Description
<code>point</code>	A point in either a geographic or cartesian coordinate system.
<code>lowerLeft</code>	A point in the same CRS as 'point'.
<code>upperRight</code>	A point in the same CRS as 'point'.

### 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.

### Example 216. `point.withinBBox()`

#### Query

```
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 460. Result

result
<code>true</code>
Rows: 1

### Example 217. point.withinBBox()

#### Query

```
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 461. Result

count(t)
1

Rows: 1

### Example 218. point.withinBBox()

#### 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 462. Result

result
true

Rows: 1



## Example 219. point.withinBBox()

### 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 463. Result

in
<null>
Rows: 1

## point() - WGS 84 2D

`point({longitude | x, latitude | y [, crs][, srid]})` returns a 2D point in the WGS 84 CRS corresponding to the given coordinate values.

### Syntax:

```
point({longitude | x, latitude | y [, crs][, srid]})
```

### Returns:

A 2D point in WGS 84.

### Arguments:

Name	Description
A single map consisting of the following:	
<code>longitude/x</code>	A numeric expression that represents the longitude/x value in decimal degrees.
<code>latitude/y</code>	A numeric expression that represents the latitude/y value in decimal degrees.
<code>crs</code>	The optional string 'WGS-84'.
<code>srid</code>	The optional number 4326.

### Considerations:

If any argument provided to `point()` is `null`, `null` will be returned.

If the coordinates are specified using `latitude` and `longitude`, the `crs` or `srid` fields are optional and inferred to be `'WGS-84'` (`srid:4326`).

If the coordinates are specified using `x` and `y`, then either the `crs` or `srid` field is required if a geographic CRS is desired.

#### Example 220. `point()`

##### 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 464. Result

point
<code>point({x: 56.7, y: 12.78, crs: 'wgs-84'})</code>
Rows: 1

#### Example 221. `point()`

##### 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 465. Result

point
<code>point({x: 2.3, y: 4.5, crs: 'wgs-84'})</code>
Rows: 1

### Example 222. point()

#### 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 466. Result

officePoint
<code>point({x: 12.994341, y: 55.611784, crs: 'wgs-84'})</code>
Rows: 1

### Example 223. point()

#### Query

```
RETURN point(null) AS p
```

If `null` is provided as the argument, `null` is returned.

#### Table 467. Result

p
<code>&lt;null&gt;</code>
Rows: 1

## point() - WGS 84 3D

`point({longitude | x, latitude | y, height | z, [, crs][, srid]})` returns a 3D point in the WGS 84 CRS corresponding to the given coordinate values.

#### Syntax:

```
point({longitude | x, latitude | y, height | z, [, crs][, srid]})
```

#### Returns:

A 3D point in WGS 84.

#### Arguments:

Name	Description
<code>A single map consisting of the following:</code>	

Name	Description
<code>longitude/x</code>	A numeric expression that represents the longitude/x value in decimal degrees.
<code>latitude/y</code>	A numeric expression that represents the latitude/y value in decimal degrees.
<code>height/z</code>	A numeric expression that represents the height/z value in meters.
<code>crs</code>	The optional string <code>'WGS-84-3D'</code> .
<code>srid</code>	The optional number <code>4979</code> .

### 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.

### Example 224. `point()`

#### Query

```
RETURN point({longitude: 56.7, latitude: 12.78, height: 8}) AS point
```

A 3D point with a `longitude` of `56.7`, a `latitude` of `12.78` and a height of `8` meters in the WGS 84 CRS is returned.

#### Table 468. Result

point
<code>point({x: 56.7, y: 12.78, z: 8.0, crs: 'wgs-84-3d'})</code>
Rows: 1

## `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:

Name	Description
A single map consisting of the following:	
<code>x</code>	A numeric expression.
<code>y</code>	A numeric expression.
<code>crs</code>	The optional string <code>'cartesian'</code> .
<code>srid</code>	The optional number <code>7203</code> .

### 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`).

### Example 225. `point()`

#### 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 469. Result

point
<code>point({x: 2.3, y: 4.5, crs: 'cartesian'})</code>
Rows: 1

## `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:

Name	Description
A single map consisting of the following:	
x	A numeric expression.
y	A numeric expression.
z	A numeric expression.
crs	The optional string 'cartesian-3D'.
srid	The optional number 9157.

### Considerations:

If any argument provided to `point()` is `null`, `null` will be returned.

If the `z` key and value is not provided, a 2D point in the Cartesian CRS will be returned.

The `crs` or `srid` fields are optional and default to the 3D Cartesian CRS (which means `srid:9157`).

### Example 226. `point()`

#### 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 470. Result

point
<code>point({x: 2.3, y: 4.5, z: 2.0, crs: 'cartesian-3d'})</code>

Rows: 1

## 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()`

## 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.

## 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.

## 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:

Type	Description	Usage	Developing
Scalar	For each row the function takes parameters and returns a result.	<a href="#">Using UDF</a>	<a href="#">Extending Neo4j (UDF)</a>

Type	Description	Usage	Developing
Aggregating	Consumes many rows and produces an aggregated result.	<a href="#">Using aggregating UDF</a>	<a href="#">Extending Neo4j (Aggregating UDF)</a>

## User-defined scalar functions

For each incoming row the function takes parameters and returns a single result.

For developing and deploying user-defined functions in Neo4j, see [Extending Neo4j → User-defined functions](#).

*Example 227. Call a user-defined function*

This example shows how you invoke a user-defined function called `join` from Cypher.

This calls the user-defined function `org.neo4j.procedure.example.join()`.

Query

```
MATCH (n:Member)
RETURN org.neo4j.procedure.example.join(collect(n.name)) AS members
```

Table 471. Result

members
"John,Paul,George,Ringo"
Rows: 1

## User-defined aggregation functions

Aggregating functions consume many rows and produces a single aggregated result.



### Example 228. Call a user-defined aggregation function

This example shows how you invoke a user-defined aggregation function called `longestString` from Cypher.

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 472. Result

member
"George"

Rows: 1

# Indexes for search performance

*This section explains how to manage indexes used for search performance.*

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](#).

## 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.

# Syntax



The index name must be unique among both indexes and constraints.



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 `CREATE ... INDEX ...` 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.



More details about the syntax descriptions can be found [here](#).

Table 473. Create a single-property index on nodes Deprecated

Syntax	<pre>CREATE [BTREE] INDEX [index_name] [IF NOT EXISTS] FOR (n:LabelName) ON (n.propertyName) [OPTIONS "{" option: value[, ...] "}"]</pre>
Description	Create a single-property index on nodes.  Index provider and configuration can be specified using the <code>OPTIONS</code> clause.
Note	Explicit use of <code>BTREE</code> keyword or B-tree options are deprecated in 4.4 and will be replaced in 5.0.

Table 474. Create a single-property index on relationships Deprecated

Syntax	<pre>CREATE [BTREE] INDEX [index_name] [IF NOT EXISTS] FOR ()-["r:TYPE_NAME"]-() ON (r.propertyName) [OPTIONS "{" option: value[, ...] "}"]</pre>
Description	Create a single-property index on relationships.  Index provider and configuration can be specified using the <code>OPTIONS</code> clause.
Note	Explicit use of <code>BTREE</code> keyword or B-tree options are deprecated in 4.4 and will be replaced in 5.0.

Table 475. Create a composite index on nodes Deprecated

Syntax	<pre>CREATE [BTREE] INDEX [index_name] [IF NOT EXISTS] FOR (n:LabelName) ON (n.propertyName_1,     n.propertyName_2,     ...     n.propertyName_n) [OPTIONS "{" option: value[, ...] "}"]</pre>
Description	<p>Create a composite index on nodes.</p> <p>Index provider and configuration can be specified using the <b>OPTIONS</b> clause.</p>
Note	<p>Explicit use of <b>BTREE</b> keyword or B-tree options are deprecated in 4.4 and will be replaced in 5.0.</p>

Table 476. Create a composite index on relationships Deprecated

Syntax	<pre>CREATE [BTREE] INDEX [index_name] [IF NOT EXISTS] FOR ()-"[r:TYPE_NAME]"-( ) ON (r.propertyName_1,     r.propertyName_2,     ...     r.propertyName_n) [OPTIONS "{" option: value[, ...] "}"]</pre>
Description	<p>Create a composite index on relationships.</p> <p>Index provider and configuration can be specified using the <b>OPTIONS</b> clause.</p>
Note	<p>Explicit use of <b>BTREE</b> keyword or B-tree options are deprecated in 4.4 and will be replaced in 5.0.</p>

Table 477. Create a node label lookup index

Syntax	<pre>CREATE LOOKUP INDEX [index_name] [IF NOT EXISTS] FOR (n) ON EACH labels(n) [OPTIONS "{" option: value[, ...] "}"]</pre>
Description	<p>Create a node label lookup index.</p> <p>Index provider can be specified using the <b>OPTIONS</b> clause.</p>

Table 478. Create a relationship type lookup index

Syntax	<pre>CREATE LOOKUP INDEX [index_name] [IF NOT EXISTS] FOR ()-["r"]-() ON [EACH] type(r) [OPTIONS "{" option: value[, ...] "}"]</pre>
Description	<p>Create a relationship type lookup index.</p> <p>Index provider can be specified using the <b>OPTIONS</b> clause.</p>

Table 479. Create a text index on nodes

Syntax	<pre>CREATE TEXT INDEX [index_name] [IF NOT EXISTS] FOR (n:LabelName) ON (n.propertyName) [OPTIONS "{" option: value[, ...] "}"]</pre>
Description	<p>Create a text index on nodes where the property has a string value.</p> <p>Index provider can be specified using the <b>OPTIONS</b> clause.</p>

Table 480. Create a text index on relationships

Syntax	<pre>CREATE TEXT INDEX [index_name] [IF NOT EXISTS] FOR ()-["r:TYPE_NAME"]-() ON (r.propertyName) [OPTIONS "{" option: value[, ...] "}"]</pre>
Description	<p>Create a text index on relationships where the property has a string value.</p> <p>Index provider can be specified using the <b>OPTIONS</b> clause.</p>

Table 481. Drop an index

Syntax	<pre>DROP INDEX index_name [IF EXISTS]</pre>
Description	<p>Drop an index of any index type.</p>
Note	<p>The command is optionally idempotent, with the default behavior to throw an error if you attempt to drop the same index twice. With <b>IF EXISTS</b>, no error is thrown and nothing happens should the index not exist.</p>

Table 482. Drop a single-property index **Deprecated**

Syntax	<pre>DROP INDEX ON :LabelName(propertyName)</pre>
--------	---

Description	Drop a single-property index on nodes without specifying a name.
Note	This syntax is deprecated.

Table 483. Drop a composite index Deprecated

Syntax	<pre>DROP INDEX ON :LabelName (n.propertyName_1, n.propertyName_2, ... n.propertyName_n)</pre>
Description	Drop a composite index on nodes without specifying a name.
Note	This syntax is deprecated.

Table 484. List indexes

Syntax	<pre>SHOW [ALL   BTREE   FULLTEXT   LOOKUP   TEXT] INDEX[ES] [YIELD { *   field[, ...] } [ORDER BY field[, ...]] [SKIP n] [LIMIT n]] [WHERE expression] [RETURN field[, ...] [ORDER BY field[, ...]] [SKIP n] [LIMIT n]]</pre>
Description	List indexes in the database, either all or filtered on index type.
Note	When using the <b>RETURN</b> clause, the <b>YIELD</b> clause is mandatory and must not be omitted.

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).

## 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 IN list`
- existence check: `n.prop IS NOT NULL`
- range search: `n.prop > value`
- prefix search: `STARTS WITH`
- suffix search: `ENDS WITH`

- substring search: **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 **:Label(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 **:Label(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 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.

## CREATE INDEX examples

- [Create a single-property B-tree index for nodes](#)

- Create a single-property B-tree index for relationships
- Create a single-property B-tree index only if it does not already exist
- Create a single-property B-tree index with specified index provider **Deprecated**
- Create a single-property B-tree index with specified index configuration **Deprecated**
- Create a composite B-tree index for nodes
- Create a composite B-tree index for relationships
- Create a composite B-tree index with specified index provider and configuration **Deprecated**
- Create a node label lookup index
- Create a relationship type lookup index
- Create a token lookup index specifying the index provider
- Create a node text index
- Create a relationship text index
- Create a text index only if it does not already exist
- Create a text index specifying the index provider
- Failure to create an already existing index
- Failure to create an index with the same name as an already existing index
- Failure to create an index when a constraint already exists
- Failure to create an index with the same name as an already existing constraint

## 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 [BTREE] INDEX index_name FOR (n:Label) ON (n.property)
```



The index is not immediately available, but is created in the background.

### Example 229. CREATE BTREE INDEX

#### Query

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

Note that the index name must be unique.

#### Result

```
+-----+
| No data returned. |
+-----+
Indexes added: 1
```



## 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 [BTREE] INDEX index_name FOR ()-[r:TYPE]-() ON (r.property)
```



The index is not immediately available, but is created in the background.

### Example 230. CREATE BTREE INDEX

#### Query

```
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
```

## 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.

### Example 231. CREATE BTREE INDEX

#### Query

```
CREATE INDEX node_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.

#### Result

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

## Create a single-property B-tree index with specified index provider

**Deprecated**

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**.

## Example 232. CREATE BTREE INDEX

### 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

```
+-----+
| No data returned. |
+-----+
Indexes added: 1
```

## Create a single-property B-tree index with specified index configuration Deprecated

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.

### Example 233. CREATE BTREE INDEX

#### Query

```
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
```

## 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.



The composite index is not immediately available, but is created in the background.

### Example 234. CREATE BTREE INDEX

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

```
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
```

## 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.

### Example 235. CREATE BTREE INDEX

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

```
CREATE INDEX 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
```

## Create a composite B-tree index with specified index provider and configuration Deprecated

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.

### Example 236. CREATE BTREE INDEX

#### 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

```
+-----+
| No data returned. |
+-----+
Indexes added: 1
```

## 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)
```



The index is not immediately available, but is created in the background.

### Example 237. CREATE LOOKUP INDEX

#### Query

```
CREATE LOOKUP INDEX node_label_lookup_index FOR (n) ON EACH labels(n)
```

Note that it can only be created once and that the index name must be unique.

#### Result

```
+-----+
| No data returned. |
+-----+
Indexes added: 1
```

## 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)
```



The index is not immediately available, but is created in the background.

#### Example 238. CREATE LOOKUP INDEX

##### 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

```
+-----+
| No data returned. |
+-----+
Indexes added: 1
```

## 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.

#### Example 239. CREATE LOOKUP INDEX

##### 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

```
+-----+
| No data returned. |
+-----+
Indexes added: 1
```

## Create a node text index

A named text index on a single property for all nodes with a particular label can be created with: Text indexes only recognize string values, do not support multiple properties, and that the index name must be unique.

```
CREATE TEXT INDEX index_name FOR (n:Label) ON (n.property)
```



The index is not immediately available, but is created in the background.

### Example 240. CREATE TEXT INDEX

#### Query

```
CREATE TEXT INDEX node_index_name FOR (n:Person) ON (n.nickname)
```

#### Result

```
+-----+
| No data returned. |
+-----+
Indexes added: 1
```

## 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)
```



The index is not immediately available, but is created in the background.

### Example 241. CREATE TEXT INDEX

#### 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
```

## 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.

### Example 242. CREATE TEXT INDEX

#### 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.

#### Result

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

## 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.

### Example 243. CREATE TEXT INDEX

#### Query

```
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

```
+-----+  
| No data returned. |  
+-----+  
Indexes added: 1
```

## 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.



#### Example 244. CREATE BTREE INDEX

##### Query

```
CREATE INDEX bookTitleIndex FOR (book:Book) ON (book.title)
```

In this case the index can not be created because it already exists.

##### Error message

```
There already exists an index (:Book {title}).
```

## 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.

#### Example 245. CREATE BTREE INDEX

##### Query

```
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'.
```

## 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.

#### Example 246. CREATE BTREE INDEX

##### Query

```
CREATE INDEX bookIsbnIndex FOR (book:Book) ON (book.isbn)
```

In this case the index can not 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.
```

## 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.

Example 247. CREATE BTREE INDEX

### Query

```
CREATE INDEX bookRecommendations FOR (book:Book) ON (book.recommendations)
```

In this case the index can not be created because there already exists a constraint with that name.

### Error message

```
There already exists a constraint called 'bookRecommendations'.
```

## SHOW INDEXES

Listing indexes can be done with `SHOW INDEXES`, which will produce a table with the following columns:



The command `SHOW INDEXES` returns only the default output. For a full output use the optional `YIELD` command. Full output: `SHOW INDEXES YIELD *`.

Table 485. List indexes output

Column	Description
<code>id</code>	The id of the index. <a href="#">Default output</a>
<code>name</code>	Name of the index (explicitly set by the user or automatically assigned). <a href="#">Default output</a>
<code>state</code>	Current state of the index. <a href="#">Default output</a>
<code>populationPercent</code>	% of index population. <a href="#">Default output</a>
<code>uniqueness</code>	Tells if the index is only meant to allow one value per key. <a href="#">Default output</a>
<code>type</code>	The IndexType of this index ( <code>BTREE</code> , <code>FULLTEXT</code> , <code>LOOKUP</code> , or <code>TEXT</code> ). <a href="#">Default output</a>
<code>entityType</code>	Type of entities this index represents (nodes or relationship). <a href="#">Default output</a>
<code>labelsOrTypes</code>	The labels or relationship types of this index. <a href="#">Default output</a>
<code>properties</code>	The properties of this index. <a href="#">Default output</a>
<code>indexProvider</code>	The index provider for this index. <a href="#">Default output</a>
<code>options</code>	The options passed to <code>CREATE</code> command.

Column	Description
<code>failureMessage</code>	The failure description of a failed index.
<code>createStatement</code>	Statement used to create the index.

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.

## SHOW INDEXES examples

- [Listing all indexes](#)
- [Listing indexes with filtering](#)

### 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 *`.

Example 248. `SHOW INDEXES`

#### Query

```
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 |
|-----|-----|-----|-----|-----|-----|-----|
| 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" |
| 5 | "index_763f72db" | "ONLINE" | 100.0             | "NONUNIQUE" | "TEXT" | "NODE"     |
["Person"] | ["middlename"] | "text-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" |
| 6 | "index_eadb868e" | "ONLINE" | 100.0             | "NONUNIQUE" | "TEXT" | "NODE"     |
["Person"] | ["surname"] | "text-1.0" |
+-----+
6 rows
```

## 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.

### Example 249. SHOW BTREE INDEXES

#### Query

```
SHOW BTREE INDEXES WHERE uniqueness = 'NONUNIQUE'
```

This will only return the default output columns. To get all columns, use `SHOW INDEXES YIELD * WHERE ...`

#### Result

```
+-----+
+-----+
| id | name                | state      | populationPercent | uniqueness | type   | entityType |
|-----|-----|-----|-----|-----|-----|-----|
| 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_deeafb2" | "ONLINE" | 100.0             | "NONUNIQUE" | "BTREE" | "NODE"      |
["Person"] | ["firstname"] | "native-btree-1.0" |
+-----+
5 rows
```

## DROP INDEX

An index can be dropped (removed) 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`.

## DROP INDEX examples

- [Drop an index](#)
- [Drop a non-existing index](#)

## Drop an index

### Example 250. DROP INDEX

#### Query

```
DROP INDEX index_name
```

#### Result

```
+-----+  
| No data returned. |  
+-----+  
Indexes removed: 1
```

## 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 **IF EXISTS**.

### Example 251. DROP INDEX

#### Query

```
DROP INDEX missing_index_name IF EXISTS
```

#### Result

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

## 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.

## Syntax




	The index name must be unique among both indexes and constraints.
	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 <code>CREATE ... INDEX ...</code> command is optionally idempotent, with the default behavior to throw an error if you attempt to create the same index twice. With <code>IF NOT EXISTS</code> , 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.

Table 486. Create a range index on nodes

Syntax	<pre>CREATE RANGE INDEX [index_name] [IF NOT EXISTS] FOR (n:LabelName) ON (n.propertyName_1[,     n.propertyName_2,     ...     n.propertyName_n]) [OPTIONS {" option: value[, ...] }"]</pre>
Description	Create a range index on nodes, either on a single property or composite.  Index provider can be specified using the <code>OPTIONS</code> clause.

Table 487. Create a range index on relationships

Syntax	<pre>CREATE RANGE INDEX [index_name] [IF NOT EXISTS] FOR ()-"[r:TYPE_NAME]"-() ON (r.propertyName_1[,     r.propertyName_2,     ...     r.propertyName_n]) [OPTIONS {" option: value[, ...] }"]</pre>
Description	Create a range index on relationships, either on a single property or composite.  Index provider can be specified using the <code>OPTIONS</code> clause.

Table 488. Create a point index on nodes

Syntax	<pre>CREATE POINT INDEX [index_name] [IF NOT EXISTS] FOR (n:LabelName) ON (n.propertyName) [OPTIONS "{" option: value[, ...] "}"]</pre>
Description	<p>Create a point index on nodes where the property has a point value.</p> <p>Index provider and configuration can be specified using the <b>OPTIONS</b> clause.</p>

Table 489. Create a point index on relationships

Syntax	<pre>CREATE POINT INDEX [index_name] [IF NOT EXISTS] FOR ()-"[r:TYPE_NAME]"-( ) ON (r.propertyName) [OPTIONS "{" option: value[, ...] "}"]</pre>
Description	<p>Create a point index on relationships where the property has a point value.</p> <p>Index provider and configuration can be specified using the <b>OPTIONS</b> clause.</p>

Table 490. Drop an index

Syntax	<pre>DROP INDEX index_name [IF EXISTS]</pre>
Description	<p>Drop an index of any index type.</p> <p>This is the <b>same command</b> as for the existing indexes.</p>
Note	<p>The command is optionally idempotent, with the default behavior to throw an error if you attempt to drop the same index twice. With <b>IF EXISTS</b>, no error is thrown and nothing happens should the index not exist.</p>

Table 491. List indexes

Syntax	<pre>SHOW [ALL   BTREE   FULLTEXT   LOOKUP   POINT   RANGE   TEXT] INDEX[ES] [YIELD { *   field[, ...] } [ORDER BY field[, ...]] [SKIP n] [LIMIT n]] [WHERE expression] [RETURN field[, ...] [ORDER BY field[, ...]] [SKIP n] [LIMIT n]]</pre>
Description	<p>Extending the existing <b>list index command</b> with filters for the new types.</p>
Note	<p>When using the <b>RETURN</b> clause, the <b>YIELD</b> clause is mandatory and must not be omitted.</p>

## Examples

- [Create a single-property range index for nodes](#)
- [Create a single-property range index for relationships](#)
- [Create a range index only if it does not already exist](#)
- [Create a range index specifying the index provider](#)
- [Create a composite range index for nodes](#)
- [Create a composite range index for relationships](#)
- [Create a node point index](#)
- [Create a relationship point index](#)
- [Create a point index only if it does not already exist](#)
- [Create a point index specifying the index provider](#)
- [Create a point index specifying the index configuration](#)
- [Create a point index specifying both the index provider and configuration](#)

### 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)
```



The index is not immediately available, but is created in the background.

#### Example 252. CREATE RANGE INDEX

##### 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
```

### 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)
```



The index is not immediately available, but is created in the background.

#### Example 253. CREATE RANGE INDEX

##### 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
```

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.

#### Example 254. CREATE RANGE INDEX

##### 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.

##### Result

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

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.

### Example 255. CREATE RANGE INDEX

#### 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
```

## 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.



The composite index is not immediately available, but is created in the background.

### Example 256. CREATE RANGE INDEX

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
```

## 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.



The composite index is not immediately available, but is created in the background.

#### Example 257. CREATE RANGE INDEX

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
```

## 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)
```



The index is not immediately available, but is created in the background.

#### Example 258. CREATE POINT INDEX

##### 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
```

## 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)
```



The index is not immediately available, but is created in the background.

### Example 259. CREATE POINT INDEX

#### 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.

#### Result

```
+-----+
| No data returned. |
+-----+
Indexes added: 1
```

## 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.

### Example 260. CREATE POINT INDEX

#### 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. |
+-----+
```

## 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.

## Example 261. CREATE POINT INDEX

### 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
```

## 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`
- `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.

## Example 262. CREATE POINT INDEX

### 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]
  }
}
```

Can be combined with specifying index provider.

### Result

```
+-----+
| No data returned. |
+-----+
Indexes added: 1
```

## 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.

## Example 263. CREATE POINT INDEX

### 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

```
+-----+
| No data returned. |
+-----+
Indexes added: 1
```

## Deprecated syntax Deprecated



This syntax only supports dropping B-tree node property indexes, all others can only be dropped by name.

## 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)
```

## Example 264. DROP INDEX

### Query

```
DROP INDEX ON :Person(firstname)
```

### Result

```
+-----+
| No data returned. |
+-----+
Indexes removed: 1
```

## 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)
```

### Example 265. DROP INDEX

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
```



# 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](#).

## 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:

Usage	Procedure/Command	Description
Create full-text node index.	<code>CREATE FULLTEXT INDEX ...</code>	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 <code>db.index.fulltext.listAvailableAnalyzers</code> 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.
Create full-text relationship index.	<code>CREATE FULLTEXT INDEX ...</code>	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 <code>db.index.fulltext.listAvailableAnalyzers</code> 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.
List available analyzers.	<code>db.index.fulltext.listAvailableAnalyzers</code>	List the available analyzers that the full-text indexes can be configured with.

Usage	Procedure/Command	Description
Use full-text node index.	<code>db.index.fulltext.queryNodes</code>	Query the given full-text index. Returns the matching nodes and their Lucene query score, ordered by score.
Use full-text relationship index.	<code>db.index.fulltext.queryRelationships</code>	Query the given full-text index. Returns the matching relationships and their Lucene query score, ordered by score.
Drop full-text index.	<code>DROP INDEX ...</code>	Drop the specified index.
Eventually consistent indexes.	<code>db.index.fulltext.awaitEventuallyConsistentIndexRefresh</code>	Wait for the updates from recently committed transactions to be applied to any eventually-consistent full-text indexes.
Listing all fulltext indexes.	<code>SHOW FULLTEXT INDEXES</code>	Lists all fulltext indexes, see <a href="#">the SHOW INDEXES command</a> for details.

## 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.



More details about the syntax descriptions can be found [here](#).

Table 492. Syntax for creating fulltext indexes

Command	Description	Comment
<pre>CREATE FULLTEXT INDEX [index_name] [IF NOT EXISTS] FOR (n:LabelName[" " ...]) ON EACH [" n.propertyName[, ...] "]" [OPTIONS "{" option: value[, ...] "}"]</pre>	<p>Create a fulltext index on nodes.</p>	<p>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.</p> <p>The index name must be unique among all indexes and constraints.</p> <p>Index provider and configuration can be specified using the <b>OPTIONS</b> clause.</p>
<pre>CREATE FULLTEXT INDEX [index_name] [IF NOT EXISTS] FOR ()-["r:TYPE_NAME[" " ...]"]-() ON EACH [" r.propertyName[, ...] "]" [OPTIONS "{" option: value[, ...] "}"]</pre>	<p>Create a fulltext index on relationships.</p>	<p>The command is optionally idempotent, with the default behavior to throw an error if you attempt to create the same index twice. With <b>IF NOT EXISTS</b>, 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.</p>

## Example 266. CREATE FULLTEXT INDEX

For instance, if we have a movie with a title.

Query

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

Table 493. Result

m.title
"The Matrix"

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

```
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

```
CALL db.index.fulltext.queryNodes("titlesAndDescriptions", "matrix") YIELD node, score  
RETURN node.title, node.description, score
```

Table 494. Result

node.title	node.description	score
"The Matrix"	<null>	0.7799721956253052

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.

### Example 267. CREATE FULLTEXT INDEX

#### Query

```
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 495. Result

(empty result)

Rows: 0

Indexes added: 1

## 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.

### Example 268. Query full-text

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 496. Result

node.title	score
"Full Metal Jacket"	1.411118507385254
"Full Moon High"	0.44524085521698
"Yellow Jacket"	0.3509605824947357
"The Jacket"	0.3509605824947357

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.

### Example 269. Query full-text

#### 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 497. Result

node.title	score
"Full Metal Jacket"	1.411118507385254

Rows: 1

Lucene also allows us to use logical operators, such as **AND** and **OR**, to search for terms.

### Example 270. Query full-text

#### Query

```
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 498. Result

node.title	score
"Full Metal Jacket"	1.1113792657852173
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.

### Example 271. Query full-text

#### Query

```
CALL db.index.fulltext.queryNodes("titlesAndDescriptions", 'description:"surreal adventure"') YIELD
node, score
RETURN node.title, node.description, score
```

#### Table 499. Result

node.title	node.description	score
"Metallica Through The Never"	"The movie follows the young roadie Trip through his surreal adventure with the band."	0.2615291476249695
Rows: 1		

A complete description of the Lucene query syntax can be found in the [Lucene documentation](#).

## Drop full-text indexes

A full-text node index is dropped by using the [same command as for other indexes](#), **DROP INDEX**.



## Example 272. DROP INDEX

In the following example, we will drop the `taggedByRelationshipIndex` that we created previously:

Query

```
DROP INDEX taggedByRelationshipIndex
```

Table 500. Result

(empty result)

Rows: 0

Indexes removed: 1

# Constraints

This section explains how to manage constraints used for ensuring data integrity.

## 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 Enterprise edition

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 Enterprise edition

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 Enterprise edition

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.

# 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 DBMS 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.

## Syntax



More details about the syntax descriptions can be found [here](#).

## 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[, ...] "}"]
```

```
CREATE CONSTRAINT [constraint_name] [IF NOT EXISTS]
FOR (n:LabelName)
REQUIRE (n.propertyName_1, ..., n.propertyName_n) IS UNIQUE
[OPTIONS "{" option: value[, ...] "}"]
```

Index provider and configuration can be specified using the `OPTIONS` clause.

## Create a node property existence constraint Enterprise edition

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 Enterprise edition

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 Enterprise edition

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.

## Syntax for dropping constraints

### Drop a constraint

The preferred way of dropping a constraint is by the name of the constraint.

```
DROP CONSTRAINT constraint_name [IF EXISTS]
```

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 a unique constraint without specifying a name Deprecated

An old way of dropping a uniqueness constraint was to drop the constraint by specifying the schema of the constraint.

```
DROP CONSTRAINT
ON (n:LabelName)
ASSERT n.propertyName IS UNIQUE
```

```
DROP CONSTRAINT
ON (n:LabelName)
ASSERT (n.propertyName_1, ..., n.propertyName_n) IS UNIQUE
```

Drop a node property existence constraint without specifying a name

Deprecated

An old way of dropping a node property existence constraint was to drop the constraint by specifying the schema of the constraint.

```
DROP CONSTRAINT
ON (n:LabelName)
ASSERT EXISTS (n.propertyName)
```

Drop a relationship property existence constraint without specifying a name

Deprecated

An old way of dropping a relationship property existence constraint was to drop the constraint by specifying the schema of the constraint.

```
DROP CONSTRAINT
ON ()-"["r:RELATIONSHIP_TYPE"]"-()
ASSERT EXISTS (r.propertyName)
```

Drop a node key constraint without specifying a name **Deprecated**

An old way of dropping a node key constraint was to drop the constraint by specifying the schema of the constraint.

```
DROP CONSTRAINT
ON (n:LabelName)
ASSERT n.propertyName IS NODE KEY
```

```
DROP CONSTRAINT
ON (n:LabelName)
ASSERT (n.propertyName_1, ..., n.propertyName_n) IS NODE KEY
```

## Syntax for listing constraints

List constraints in the database, either all or filtered on constraint type. This requires the **SHOW CONSTRAINT** privilege.

The simple version of the command allows for a **WHERE** clause and will give back the default set of output columns:

```
SHOW [ALL
      |UNIQUE
      |NODE [PROPERTY] EXIST[ENCE]
      |REL[ATIONSHIP] [PROPERTY] EXIST[ENCE]
      |[PROPERTY] EXIST[ENCE]
      |NODE KEY] CONSTRAINT[S]
      [WHERE expression]
```

To get the full set of output columns, a yield clause is needed:

```
SHOW [ALL
      |UNIQUE
      |NODE [PROPERTY] EXIST[ENCE]
      |REL[ATIONSHIP] [PROPERTY] EXIST[ENCE]
      |[PROPERTY] EXIST[ENCE]
      |NODE KEY] CONSTRAINT[S]
      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 501. Listing constraints output

Column	Description
<code>id</code>	The id of the constraint. <a href="#">Default output</a>
<code>name</code>	Name of the constraint (explicitly set by the user or automatically assigned). <a href="#">Default output</a>
<code>type</code>	The ConstraintType of this constraint ( <code>UNIQUENESS</code> , <code>NODE_PROPERTY_EXISTENCE</code> , <code>NODE_KEY</code> , or <code>RELATIONSHIP_PROPERTY_EXISTENCE</code> ). <a href="#">Default output</a>
<code>entityType</code>	Type of entities this constraint represents (nodes or relationship). <a href="#">Default output</a>
<code>labelsOrTypes</code>	The labels or relationship types of this constraint. <a href="#">Default output</a>
<code>properties</code>	The properties of this constraint. <a href="#">Default output</a>
<code>ownedIndexId</code>	The id of the index associated to the constraint, or <code>null</code> if no index is associated with the constraint. <a href="#">Default output</a>

Column	Description
<code>options</code>	The options passed to <code>CREATE</code> command, for the index associated to the constraint, or <code>null</code> if no index is associated with the constraint.
<code>createStatement</code>	Statement used to create the constraint.



The deprecated built-in procedures for listing constraints, such as `db.constraints`, work as before and are not affected by the `SHOW CONSTRAINTS` privilege.

## Examples

*Examples of how to manage constraints used for ensuring data integrity.*

### Unique node property constraints

- [Create a unique constraint](#)
- [Create a unique constraint only if it does not already exist](#)
- [Create a unique constraint with specified index provider and configuration](#)
- [Failure to create an already existing unique property constraint](#)
- [Failure to create a unique property constraint on same schema as existing index](#)
- [Create a node that complies with unique property constraints](#)
- [Create a node that violates a unique property constraint](#)
- [Failure to create a unique property constraint due to conflicting nodes](#)

### 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.



### Example 273. CREATE CONSTRAINT

#### 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.



No constraint will be created if any other constraint with that name or another uniqueness constraint on the same schema already exists.

### Example 274. CREATE CONSTRAINT

#### Query

```
CREATE CONSTRAINT constraint_name IF NOT EXISTS
FOR (book:Book) REQUIRE book.isbn IS UNIQUE
```

#### 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 Neo4j 4.4, B-tree index-backed constraints are still the correct alternative to use.

#### Example 275. CREATE CONSTRAINT

##### Query

```
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

#### Example 276. CREATE CONSTRAINT

Create a unique property constraint on the property `title` on nodes with the `Book` label, when that constraint already exists.

##### Query

```
CREATE CONSTRAINT FOR (book:Book) REQUIRE book.title IS UNIQUE
```

In this case the constraint can not 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

### Example 277. CREATE CONSTRAINT

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 not 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

### Example 278. CREATE CONSTRAINT

Create a `Book` node with an `isbn` that is not 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

### Example 279. CREATE 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 is not created in the graph.

Error message

```
Node(0) already exists with label `Book` and property `isbn` = '1449356265'
```

## Failure to create a unique property constraint due to conflicting nodes

### Example 280. CREATE CONSTRAINT

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 not be created because it is violated by existing data. You 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'
```

## Node property existence constraints Enterprise edition

- [Create a node property existence constraint](#)
- [Create a node property existence constraint only if it does not already exist](#)
- [Failure to create an already existing node property existence constraint](#)
- [Create a node that complies with property existence constraints](#)
- [Create a node that violates a property existence constraint](#)
- [Removing an existence constrained node property](#)
- [Failure to create a node property existence constraint due to existing node](#)

## 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.

#### Example 281. CREATE CONSTRAINT

##### 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. No constraint will be created if any other constraint with that name or another node property existence constraint on the same schema already exists.

#### Example 282. CREATE CONSTRAINT

##### Query

```
CREATE CONSTRAINT constraint_name IF NOT EXISTS
FOR (book:Book) REQUIRE book.isbn IS NOT NULL
```

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

### Example 283. CREATE CONSTRAINT

Create a node property existence constraint on the property `title` on nodes with the `Book` label, when that constraint already exists.

#### Query

```
CREATE CONSTRAINT booksShouldHaveTitles
FOR (book:Book) REQUIRE book.title IS NOT NULL
```

In this case the constraint can not be created because it already exists.

#### Error message

```
Constraint already exists:
Constraint( id=3, name='preExistingNodePropExist', type='NODE PROPERTY EXISTENCE', schema=(Book
{title}) )
```

## Create a node that complies with property existence constraints

### Example 284. CREATE CONSTRAINT

Create a `Book` node with an `isbn` property.

#### Query

```
CREATE (book:Book {isbn: '1449356265', title: 'Graph Databases'})
```

#### Result

```
+-----+
| No data returned. |
+-----+
Nodes created: 1
Properties set: 2
Labels added: 1
```

## Create a node that violates a property existence constraint

### Example 285. CREATE CONSTRAINT

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

Query

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

In this case the node is not created in the graph.

Error message

```
Node(0) with label `Book` must have the property `isbn`
```

### Removing an existence constrained node property

#### Example 286. CREATE CONSTRAINT

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

### Example 287. CREATE CONSTRAINT

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`
```

## Relationship property existence constraints Enterprise edition

- [Create a relationship property existence constraint](#)
- [Create a relationship property existence constraint only if it does not already exist](#)
- [Failure to create an already existing relationship property existence constraint](#)
- [Create a relationship that complies with property existence constraints](#)
- [Create a relationship that violates a property existence constraint](#)
- [Removing an existence constrained relationship property](#)
- [Failure to create a relationship property existence constraint due to existing relationship](#)

### 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.

### Example 288. CREATE CONSTRAINT

#### 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. No constraint will be created if any other constraint with that name or another relationship property existence constraint on the same schema already exists.

### Example 289. CREATE CONSTRAINT

#### Query

```
CREATE CONSTRAINT constraint_name
IF NOT EXISTS FOR ()-[like:LIKED]-() REQUIRE like.day IS NOT NULL
```

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

### Example 290. CREATE 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 not be created because there already exists a constraint with that name.

#### Error message

```
There already exists a constraint called 'relPropExist'.
```

## Create a relationship that complies with property existence constraints

### Example 291. CREATE CONSTRAINT

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

#### Example 292. CREATE 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 is not created in the graph.

Error message

```
Relationship(0) with type `LIKED` must have the property `day`
```

### Removing an existence constrained relationship property

### Example 293. CREATE CONSTRAINT

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

### Example 294. CREATE CONSTRAINT

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 not 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`
```

## Node key constraints Enterprise edition

- [Create a node key constraint](#)
- [Create a node key constraint only if it does not already exist](#)
- [Create a node key constraint with specified index provider](#)
- [Create a node key constraint with specified index configuration](#)
- [Failure to create a node key constraint when a unique property constraint exists on the same schema](#)
- [Failure to create a node key constraint with the same name as existing index](#)
- [Create a node that complies with node key constraints](#)

- [Create a node that violates a node key constraint](#)
- [Removing a NODE KEY-constrained property](#)
- [Failure to create a node key constraint due to existing node](#)

## 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.

### Example 295. CREATE CONSTRAINT

#### 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. No constraint will be created if any other constraint with that name or another node key constraint on the same schema already exists.

### Example 296. CREATE CONSTRAINT

#### Query

```
CREATE CONSTRAINT constraint_name IF NOT EXISTS
FOR (n:Person) REQUIRE (n.firstname, n.surname) IS NODE KEY
```

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 Neo4j 4.4, B-tree index-backed constraints are still the correct alternative to use.

#### Example 297. CREATE CONSTRAINT

##### Query

```
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.

## Example 298. CREATE CONSTRAINT

### Query

```
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

## Example 299. CREATE CONSTRAINT

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 not 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

### Example 300. CREATE CONSTRAINT

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

### Example 301. CREATE CONSTRAINT

Create a `Person` node with both a `firstname` and `surname` property.

#### Query

```
CREATE (p:Person {firstname: 'John', surname: 'Wood', age: 55})
```

#### Result

```
+-----+
| No data returned. |
+-----+
Nodes created: 1
Properties set: 3
Labels added: 1
```

## Create a node that violates a node key constraint

### Example 302. CREATE 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 is not created in the graph.

Error message

```
Node(0) with label `Person` must have the properties (firstname, surname)
```

### Removing a NODE KEY-constrained property

#### Example 303. CREATE CONSTRAINT

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



### Example 304. CREATE CONSTRAINT

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 not 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=(:Person {firstname, surname}) ):
Node(0) with label `Person` must have the properties (firstname, surname)
```

## Drop a constraint by name

- [Drop a constraint](#)
- [Drop a non-existing constraint](#)

## 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`.

### Example 305. DROP CONSTRAINT

#### 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.

### Example 306. DROP CONSTRAINT

#### Query

```
DROP CONSTRAINT missing_constraint_name IF EXISTS
```

#### Result

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

## Listing constraints

- [Listing all constraints](#)
- [Listing constraints with filtering](#)

### 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 *`.



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.

### Example 307. SHOW CONSTRAINTS

#### Query

```
SHOW CONSTRAINTS
```

#### 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.

## Example 308. SHOW CONSTRAINTS

### 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

```
+-----+
+-----+
| id | name | type | entityType | labelsOrTypes |
properties | ownedIndexId |
+-----+
+-----+
| 7 | "constraint_f076a74d" | "RELATIONSHIP_PROPERTY_EXISTENCE" | "RELATIONSHIP" | ["KNOWS"] |
["since"] | <null> |
+-----+
+-----+
1 row
```

## Deprecated syntax Deprecated

- [Create a unique constraint using deprecated syntax](#)
- [Drop a unique constraint](#)
- [Create a node property existence constraint using deprecated syntax 1](#)
- [Create a node property existence constraint using deprecated syntax 2](#)
- [Drop a node property existence constraint](#)
- [Create a relationship property existence constraint using deprecated syntax 1](#)
- [Create a relationship property existence constraint using deprecated syntax 2](#)
- [Drop a relationship property existence constraint](#)
- [Create a node key constraint using deprecated syntax](#)
- [Drop a node key constraint](#)

## 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.

### Example 309. CREATE CONSTRAINT

#### 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.

### Example 310. DROP CONSTRAINT

#### 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 1

The node property existence constraint ensures that all nodes with a certain label have a certain property.

### Example 311. CREATE CONSTRAINT

#### 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.

### Example 312. CREATE CONSTRAINT

#### Query

```
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.

### Example 313. DROP CONSTRAINT

#### Query

```
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.

### Example 314. CREATE CONSTRAINT

#### Query

```
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 deprecated syntax 2

The relationship property existence constraint ensures all relationships with a certain type have a certain property.

#### Example 315. CREATE CONSTRAINT

##### Query

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

##### Result

```
+-----+
| No data returned. |
+-----+
Property existence constraints added: 1
```

## Drop a relationship property existence constraint

To remove a relationship property existence constraint from the database, use **DROP CONSTRAINT**.

#### Example 316. DROP CONSTRAINT

##### Query

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

##### Result

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

#### Example 317. CREATE CONSTRAINT

##### Query

```
CREATE CONSTRAINT ON (n:Person) ASSERT (n.firstname) IS NODE KEY
```

##### Result

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

### Example 318. DROP CONSTRAINT

#### Query

```
DROP CONSTRAINT ON (n:Person) ASSERT (n.firstname, n.surname) IS NODE KEY
```

#### Result

```
+-----+  
| No data returned. |  
+-----+  
Node key constraints removed: 1
```

# 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 section, see [Execution plans](#).

## 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.`

## Cypher version

Occasionally, there is a requirement to use a previous version of the Cypher compiler when running a query.

Here we detail the available versions:

Query option	Description	Default
<code>3.5</code>	This will force the query to use Neo4j Cypher 3.5.	
<code>4.3</code>	This will force the query to use Neo4j Cypher 4.3.	



Query option	Description	Default
4.4	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.	✓



In Neo4j 4.4, the support for Cypher 3.5 and Cypher 4.3 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.

## 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).

Option	Description	Default
<code>runtime=interpreted</code>	This will force the query planner to use the interpreted runtime.	This is not used in Enterprise Edition unless explicitly asked for. It is the only option for all queries in Community Edition—it is not necessary to specify this option in Community Edition.
<code>runtime=slotted</code>	This will cause the query planner to use the slotted runtime.	This is the default option for all queries which are not supported by <code>runtime=pipelined</code> in Enterprise Edition.
<code>runtime=pipelined</code>	This will cause the query planner to use the pipelined runtime if it supports the query. If the pipelined runtime does not support the query, the planner will fall back to the slotted runtime.	This is the default option for some queries in Enterprise Edition.


In Enterprise Edition, the Cypher query planner selects the runtime, falling back to alternative runtimes as follows:

- Try the pipelined runtime first.
- If the pipelined runtime does not support the query, then fall back to use the slotted runtime.
- Finally, if the slotted runtime does not support the query, fall back to the interpreted runtime. The interpreted runtime supports all queries, and is the only option in Neo4j Community Edition.

## 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:

Query option	Description	Default
<code>planner=cost</code>	Use cost based planning with default limits on plan search space and time.	✓
<code>planner=idp</code>	Synonym for <code>planner=cost</code> .	
<code>planner=dp</code>	Use cost based planning without limits on plan search space and time to perform an exhaustive search for the best execution plan.  <div style="border: 1px solid #ccc; padding: 5px; display: inline-block;">  Using this option can significantly increase the planning time of the query. </div>	

## 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:

Query option	Description	Default
<code>connectComponentsPlanner=greedy</code>	Use a greedy approach when combining sub-plans.  <div style="border: 1px solid #ccc; padding: 5px; display: inline-block;">  Using this option can significantly reduce the planning time of the query. </div>	
<code>connectComponentsPlanner=idp</code>	Use the cost based IDP search algorithm when combining sub-plans.  <div style="border: 1px solid #ccc; padding: 5px; display: inline-block;">  Using this option can significantly increase the planning time of the query but usually finds better plans. </div>	✓

## Cypher update strategy

This option affects the eagerness of updating queries.

The possible values are:

Query option	Description	Default
<code>updateStrategy=default</code>	Update queries are executed eagerly when needed.	✓
<code>updateStrategy=eager</code>	Update queries are always executed eagerly.	

## Cypher expression engine

This option affects how the runtime evaluates expressions.

The possible values are:

Query option	Description	Default
<code>expressionEngine=default</code>	Compile expressions and use the compiled expression engine when needed.	✓
<code>expressionEngine=interpreted</code>	Always use the <i>interpreted</i> expression engine.	
<code>expressionEngine=compiled</code>	Always compile expressions and use the <i>compiled</i> expression engine.  Cannot be used together with <code>runtime=interpreted</code> .	

## Cypher operator engine

This query option affects whether the pipelined runtime attempts to generate compiled code for groups of operators.

The possible values are:


Query option	Description	Default
<code>operatorEngine=default</code>	Attempt to generate compiled operators when applicable.	✓
<code>operatorEngine=interpreted</code>	Never attempt to generate compiled operators.	
<code>operatorEngine=compiled</code>	Always attempt to generate <i>compiled</i> operators.  Cannot be used together with <code>runtime=interpreted</code> or <code>runtime=slotted</code> .	

## Cypher interpreted pipes fallback

This query option affects how the pipelined runtime behaves for operators it does not directly support.

The available options are:

Query option	Description	Default
<code>interpretedPipesFallback=default</code>	Equivalent to <code>interpretedPipesFallback=whitelisted_plans_only</code> .	✓
<code>interpretedPipesFallback=disabled</code>	If the plan contains any operators not supported by the pipelined runtime then another runtime is chosen to execute the entire plan.  Cannot be used together with <code>runtime=interpreted</code> or <code>runtime=slotted</code> .	
<code>interpretedPipesFallback=whitelisted_plans_only</code>	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 <code>runtime=interpreted</code> or <code>runtime=slotted</code> .	
<code>interpretedPipesFallback=all</code>	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 <code>runtime=interpreted</code> or <code>runtime=slotted</code> .	

 This setting is experimental, and using it in a production environment is discouraged.

## 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.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:

Option	Description	Default
<code>replan=default</code>	This is the planning and replanning option as described above.	✓
<code>replan=force</code>	This will force a replan, even if the plan is valid according to the planning rules. Once the new plan is complete, it replaces the existing one in the query cache.	
<code>replan=skip</code>	If a valid plan already exists, it will be used even if the planning rules would normally dictate that it should be replanned.	

The replan option is prepended to queries.

For example:

```
CYPHER replan=force MATCH ...
```

In a mixed workload, you can force replanning by using the Cypher `EXPLAIN` commands. This can be useful to schedule replanning of queries which are expensive to plan, at known times of low load. Using `EXPLAIN` will make sure the query is only planned, but not executed.

For example:

```
CYPHER replan=force EXPLAIN MATCH ...
```

During times of known high load, `replan=skip` can be useful to not introduce unwanted latency spikes.

## Profile a query

There are two options to choose from when you want to analyze a query by looking at its execution plan:

### EXPLAIN

If you want to see the execution plan but not run the statement, prepend your Cypher statement with `EXPLAIN`. The statement will always return an empty result and make no changes to the database.

## PROFILE

If you want to run the statement and see which operators are doing most of the work, use **PROFILE**. This will run your statement and keep track of how many rows pass through each operator, and how much each operator needs to interact with the storage layer to retrieve the necessary data. 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.

## 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.

## 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.

The different indexes are:

- **BTREE**
- **TEXT**

## BTREE indexes

**BTREE** indexes support all types of predicates:

Predicate	Syntax
equality check	<code>n.prop = value</code>
list membership check	<code>n.prop IN list</code>
existence check	<code>n.prop IS NOT NULL</code>
range search	<code>n.prop &gt; value</code>
prefix search	<code>STARTS WITH</code>
suffix search	<code>ENDS WITH</code>
substring search	<code>CONTAINS</code>

## 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:

Predicate	Syntax
equality check	<code>n.prop = "string"</code>
list membership check	<code>n.prop IN ["a", "b", "c"]</code>
range search	<code>n.prop &gt; "string"</code>
prefix search	<code>STARTS WITH</code>
suffix search	<code>ENDS WITH</code>
substring search	<code>CONTAINS</code>

## 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.

Examples:

- Relationship **BTREE** index
- Node **TEXT** index
- Relationship **TEXT** index
- Multiple available index types
- Equality check using **WHERE** (single-property index)
- Equality check using **WHERE** (composite index)
- Range comparisons using **WHERE** (single-property index)
- Range comparisons using **WHERE** (composite index)
- Multiple range comparisons using **WHERE** (single-property index)
- Multiple range comparisons using **WHERE** (composite index)
- List membership check using **IN** (single-property index)
- List membership check using **IN** (composite index)
- Prefix search using **STARTS WITH** (single-property index)
- Prefix search using **STARTS WITH** (composite index)
- Suffix search using **ENDS WITH** (single-property index)
- Suffix search using **ENDS WITH** (composite index)
- Substring search using **CONTAINS** (single-property index)
- Substring search using **CONTAINS** (composite index)
- Existence check using **IS NOT NULL** (single-property index)
- Existence check using **IS NOT NULL** (composite index)
- Spatial distance searches (single-property index)
- Spatial distance searches (composite index)
- Spatial bounding box searches (single-property index)
- Spatial bounding box searches (composite index)

## Relationship **BTREE** index

In this example, a **KNOWS(since)** relationship **BTREE** index is available.

Query

```
MATCH (person)-[relationship:KNOWS {since: 1992}]->(friend)
RETURN person, friend
```

## Query Plan

Compiler CYPHER 4.4

Planner COST

Runtime PIPELINED

Runtime version 4.4

```
+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+
| Operator          | Details                                     | Estimated Rows |
+-----+-----+-----+-----+-----+-----+-----+
| +ProduceResults  | person, friend                             | 1              |
+-----+-----+-----+-----+-----+-----+-----+
| Pipeline 0 |
+-----+-----+-----+-----+-----+-----+-----+
| +DirectedRelationshipIndexSeek | BTREE INDEX (person)-[relationship:KNOWS(since)]->(friend) WHERE since
= $autoint_0 | 1 | 1 | 3 | 112 | 2/1 | 1.404 |
+-----+-----+-----+-----+-----+-----+-----+
| Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+-----+-----+
```

Total database accesses: 3, total allocated memory: 176

## Node TEXT index

In the example below, a **Person(surname)** node **TEXT** index is available.

### Query

```
MATCH (person:Person {surname: 'Smith'})
RETURN person
```

## Query Plan

Compiler CYPHER 4.4

Planner COST

Runtime PIPELINED

Runtime version 4.4

```
+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+
| Operator          | Details                                     | Estimated Rows |
+-----+-----+-----+-----+-----+-----+-----+
| +ProduceResults  | person                                     | 2              |
+-----+-----+-----+-----+-----+-----+-----+
| 1 | 0 | | | | Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+-----+-----+
| +NodeIndexSeek  | TEXT INDEX person:Person(surname) WHERE surname = $autostring_0 | 2              |
+-----+-----+-----+-----+-----+-----+-----+
| 1 | 2 | 112 | 2/0 | 6.367 | Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+-----+-----+
```

Total database accesses: 2, total allocated memory: 176

## Relationship TEXT index

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

```
+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
| 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 | TEXT INDEX (person)-[relationship:KNOWS(metIn)]->(friend) WHERE metIn =
$autostring_0 | 1 | 1 | 3 | 112 | 2/0 | 17.095 |
Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
```

Total database accesses: 3, total allocated memory: 176

## 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
```

## 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                                     |
1 | 1 | 0 | | | | Fused in Pipeline 0 |
| | | | | | | |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
| +NodeIndexSeek  | BTREE INDEX person:Person(middlename) WHERE middlename = $autostring_0 |
1 | 1 | 2 | 112 | 2/1 | 0.392 | Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
```

Total database accesses: 2, total allocated memory: 176

## 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

```
MATCH (person:Person)
WHERE person.firstname = 'Andy'
RETURN 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	person					1
1	0				Fused in Pipeline 0	
+NodeIndexSeek	BTREE INDEX person:Person(firstname) WHERE firstname = \$autostring_0					1
1	2	112		2/1	1.208	Fused in Pipeline 0

Total database accesses: 2, total allocated memory: 176

## 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

```
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.

### Result

```
+-----+
---+
| n
|
+-----+
---+
|
Node[0]{country:"UK",firstname:"John",highScore:54321,surname:"Smith",name:"john",middlename:"Ron",age:35}
|
+-----+
---+
1 row
```

## 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 | 1 | 1 | 0 | friend, person | | Fused in |
Pipeline 0 |
+-----+-----+-----+-----+-----+-----+-----+
| +DirectedRelationshipIndexSeekByRange | BTREE INDEX (person)-[r:KNOWS(since)]->(friend) WHERE since <
$autoint_0 | 1 | 1 | 3 | 112 | 2/1 | 1.767 | Fused
in Pipeline 0 |
+-----+-----+-----+-----+-----+-----+-----+

Total database accesses: 3, total allocated memory: 176
```

### 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
```

## 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                         | `r.since`
|           2 | 2 | 0 |           |           |           | Fused in
Pipeline 0 |
| |
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
| +Projection                             | cache[r.since] AS `r.since`
|           2 | 2 | 0 |           |           |           | Fused in
Pipeline 0 |
| |
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
| +Filter                                 | cache[r.lastMet] > $autoint_1
|           2 | 2 | 0 |           |           |           | Fused in
Pipeline 0 |
| |
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
| +UndirectedRelationshipIndexSeek | BTREE INDEX (anon_0)-[r:KNOWS(since, lastMet)]-(anon_1) WHERE since <
| $autoint_0 AND lastMet IS NOT |           2 | 2 | 3 |           112 |           1/1
|           1.181 | Fused in Pipeline 0 | | | | |
|           |           |           |           |           |           |
|           |           |           |           |           |           |
|           |           |           |           |           |           |
+-----+
+-----+
+-----+-----+-----+-----+-----+
+-----+

```

Total database accesses: 3, total allocated memory: 176

## 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

```

MATCH (person:Person)
WHERE 10000 < person.highScore < 20000
RETURN 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  | person
|           1 | 1 | 0 |           |           |           | Fused in
Pipeline 0 |
| |
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
| +NodeIndexSeekByRange | BTREE INDEX person:Person(highScore) WHERE highScore > $autoint_0 AND highScore
< $autoint_1 |           1 | 1 | 2 |           112 |           2/1 | 0.812 |
Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+
+-----+
+-----+
+-----+-----+-----+-----+-----+
+-----+

Total database accesses: 2, total allocated memory: 176
```

## 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

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
|                | 1 | 1 | 0 | | | Fused in
Pipeline 0 |
| |
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
| +NodeIndexSeek | BTREE INDEX person:Person(highScore, name) WHERE highScore > $autoint_0 AND highScore
< $autoint_1 A | 1 | 1 | 2 | 112 | 2/1 | 3.233 |
Fused in Pipeline 0 |
|                | ND name IS NOT NULL
|                | | | | | |
|
+-----+
+-----+-----+-----+-----+-----+
+-----+
+-----+
```

Total database accesses: 2, total allocated memory: 176

## List membership check using **IN** (single-property index)

The **IN** predicate on `r.lastMetIn` 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

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 | | | | | Fused in |
| Pipeline 0 | | | | | | | |
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
| +DirectedRelationshipIndexSeek | BTREE INDEX (person)-[r:KNOWS(lastMetIn)]->(friend) WHERE lastMetIn IN $autolist_0 | | | | | | |
Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
Total database accesses: 4, total allocated memory: 176
```

## 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

```
MATCH (person)-[r:KNOWS]->(friend)
WHERE r.since IN [1992, 2017] AND r.lastMet IN [2002, 2021]
RETURN person, friend
```

## 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 $autolist_0 AND lastMet IN $ |
| 1 | 1 | 6 | 112 | 5/1 |
4.788 | Fused in Pipeline 0 |
| | autolist_1 |
| | |
+-----+
+-----+-----+-----+-----+-----+
+-----+
+-----+-----+-----+-----+-----+
```

Total database accesses: 6, total allocated memory: 176

## 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
```

## 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
2 | 1 | 0 | | | | Fused in Pipeline 0 |
| |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| +NodeIndexSeekByRange | BTREE INDEX person:Person(firstname) WHERE firstname STARTS WITH $autostring_0 |
2 | 1 | 2 | 112 | 3/0 | 0.630 | Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

Total database accesses: 2, total allocated memory: 176

## 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 a 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

```
MATCH (person:Person)
WHERE person.firstname STARTS WITH 'And' AND person.surname IS NOT NULL
RETURN 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 | person
|                | 1 | 1 | 0 | | | Fused in
Pipeline 0 |
| |
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
| +NodeIndexSeek | BTREE INDEX person:Person(firstname, surname) WHERE firstname STARTS WITH
$autostring_0 AND surname | 1 | 1 | 2 | 112 | 3/0 |
0.544 | Fused in Pipeline 0 |
| | IS NOT NULL
| | | | | |
|
+-----+
+-----+-----+-----+-----+-----+
+-----+
+-----+
```

Total database accesses: 2, total allocated memory: 176

## 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
```

## 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
| 0 | 1 | 0 | | | | | Fused in
Pipeline 0 |
| |
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
| +DirectedRelationshipIndexEndsWithScan | BTREE INDEX (person)-[r:KNOWS(metIn)]->(friend) WHERE metIn
ENDS WITH $autostring_0 | 0 | 1 | 3 | 112 | 2/1 |
0.409 | Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
```

Total database accesses: 3, total allocated memory: 176

## Suffix search using **ENDS WITH** (composite index)

The **ENDS WITH** predicate on `r.metIn` in the following query uses the `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 `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

```
MATCH (person)-[r:KNOWS]->(friend)
WHERE r.metIn ENDS WITH 'mo' AND r.lastMetIn IS NOT NULL
RETURN person, friend
```



## 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
2 | 1 | 0 | | | | Fused in Pipeline 0 |
| |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
| +NodeIndexContainsScan | BTREE INDEX person:Person(firstname) WHERE firstname CONTAINS $autostring_0 |
2 | 1 | 2 | 112 | 3/0 | 1.355 | Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
```

Total database accesses: 2, total allocated memory: 176

## 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

```
MATCH (person:Person)
WHERE person.country CONTAINS '300' AND person.age IS NOT NULL
RETURN 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 | person
|           2 | 1 | 0 | | | | Fused in
Pipeline 0 |
| |
+-----+-----+-----+-----+-----+
+-----+
| +Filter         | cache[person.country] CONTAINS $autostring_0
|           2 | 1 | 0 | | | | Fused in
Pipeline 0 |
| |
+-----+-----+-----+-----+-----+
+-----+
| +NodeIndexScan | BTREE INDEX person:Person(country, age) WHERE country IS NOT NULL AND age IS NOT NULL,
cache[person. | 303 | 303 | 304 | 112 | 5/0 | 2.171 |
Fused in Pipeline 0 |
| | country]
| | | | | | |
|
+-----+
+-----+
+-----+-----+-----+-----+-----+
+-----+
+-----+

```

Total database accesses: 304, total allocated memory: 176

## 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

```

## 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 |
| |
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
| +DirectedRelationshipIndexScan | BTREE INDEX (person)-[r:KNOWS(since)]->(friend) WHERE since IS NOT NULL |
| 1 | 1 | 3 | 112 | 2/1 | 4.300 | Fused in |
Pipeline 0 |
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
Total database accesses: 3, total allocated memory: 176
```

## 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

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
|                | 1 | 2 | 0 | | | Fused in
Pipeline 0 |
| |
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
| +NodeIndexScan | BTREE INDEX p:Person(firstname, surname) WHERE firstname IS NOT NULL AND surname IS
NOT NULL | 1 | 2 | 3 | 112 | 2/1 | 2.915 | Fused
in Pipeline 0 |
+-----+
+-----+-----+-----+-----+-----+-----+
+-----+
```

Total database accesses: 3, total allocated memory: 176

## 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

```
MATCH ()-[r:KNOWS]->()
WHERE point.distance(r.lastMetPoint, point({x: 1, y: 2})) < 2
RETURN r.lastMetPoint
```

## 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                          | `r.lastMetPoint`
|      13 |    9 |    0 |                |                        |           | Fused in
Pipeline 0 |
| |
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
+-----+
| +Projection                              | cache[r.lastMetPoint] AS `r.lastMetPoint`
|      13 |    9 |    0 |                |                        |           | Fused in
Pipeline 0 |
| |
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
+-----+
| +Filter                                  | point.distance(cache[r.lastMetPoint], point({x: $autoint_0, y:
$autoint_1})) < $autoint_2
|      13 |    9 |    0 |                |                        |           | Fused in Pipeline 0 |
| |
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
+-----+
| +DirectedRelationshipIndexSeekByRange | BTREE INDEX (anon_0)-[r:KNOWS(lastMetPoint)]->(anon_1) WHERE
point.distance(lastMetPoint, point($autoint_0, $autoint_1)) < $autoint_2, cache[r.lastMetPoint]
|      13 |    9 |   19 |   112 |
5/3 |    1.596 | Fused in Pipeline 0 |
|
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
+-----+

```

Total database accesses: 19, total allocated memory: 176

## 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

```

MATCH (p:Person)
WHERE point.distance(p.place, point({x: 1, y: 2})) < 2 AND p.name IS NOT NULL
RETURN p.place

```

## 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.place`
| 0 | 9 | 0 | | | | Fused in
Pipeline 0 |
| |
+-----+-----+-----+-----+-----+-----+
| +Projection     | cache[p.place] AS `p.place`
| 0 | 9 | 0 | | | | Fused in
Pipeline 0 |
| |
+-----+-----+-----+-----+-----+-----+
| +Filter         | point.distance(cache[p.place], point({x: $autoint_0, y: $autoint_1})) < $autoint_2
| 0 | 9 | 0 | | | | Fused in
Pipeline 0 |
| |
+-----+-----+-----+-----+-----+-----+
| +NodeIndexSeek | BTREE INDEX p:Person(place, name) WHERE point.distance(place, point($autoint_0,
$autoint_1)) < $auto | 0 | 9 | 10 | 112 | 6/0 |
1.370 | Fused in Pipeline 0 |
| | int_2 AND name IS NOT NULL, cache[p.place]
| | | | | | |
|
+-----+
+-----+-----+-----+-----+-----+-----+
+-----+

Total database accesses: 10, total allocated memory: 176
```

## 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

```
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

+-----+
+-----+
+-----+-----+-----+-----+-----+
+-----+
| Operator          | Details
| Estimated Rows   | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Other
|-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
+-----+
| +ProduceResults  | `person.firstname`
| 0 | 1 | 0 | | | | Fused in
Pipeline 0 |
| |
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
| +Projection      | person.firstname AS `person.firstname`
| 0 | 1 | 2 | | | | Fused in
Pipeline 0 |
| |
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
| +NodeIndexSeekByRange | BTREE INDEX person:Person(location) WHERE point.withinBBox(location,
point($autodouble_0, $autodoubl | 0 | 1 | 2 | 112 |
6/0 | 16.182 | Fused in Pipeline 0 |
| | | | e_1), point($autodouble_2, $autodouble_3))
| | | | | | |
|
+-----+
+-----+
+-----+-----+-----+-----+-----+
+-----+

Total database accesses: 4, total allocated memory: 176

```

## 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

```

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

```
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
|                | 1 | 1 | 0 | | | Fused in
Pipeline 0 |
| |
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
| +NodeIndexSeek | BTREE INDEX person:Person(place, firstname) WHERE point.withinBBox(place,
point($autodouble_0, $auto | 1 | 1 | 2 | 112 | 6/0 |
1.065 | Fused in Pipeline 0 |
|                | double_1), point($autodouble_2, $autodouble_3)) AND firstname IS NOT NULL
|                | | | | | | |
|
+-----+
+-----+-----+-----+-----+-----+-----+
+-----+
+-----+-----+-----+-----+-----+-----+
+-----+

Total database accesses: 2, total allocated memory: 176
```

## Basic query tuning example

*This section describes how to profile a query, by using optimizations based on native index capabilities.*

Start with a basic example to help you get the hang of profiling queries. The following examples will use a movies data set.

### The data set

In this section, examples demonstrates 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:

movies.csv

```
title,released,tagline
Something's Gotta Give,1975,null
Johnny Mnemonic,1995,The hottest data on earth. In the coolest head in town
The Replacements,2000,"Pain heals, Chicks dig scars... Glory lasts forever"
The Devil's Advocate,1997,Evil has its winning ways
The Matrix Revolutions,2003,Everything that has a beginning has an end
The Matrix Reloaded,2003,Free your mind
The Matrix,1999>Welcome to the Real World
The Matrix Revolutions,2003,Everything that has a beginning has an end
The Matrix Reloaded,2003,Free your mind
The Matrix,1999>Welcome to the Real World
The Matrix Revolutions,2003,Everything that has a beginning has an end
The Matrix Reloaded,2003,Free your mind
The Matrix,1999>Welcome to the Real World
V for Vendetta,2006,Freedom! Forever!
Cloud Atlas,2012,Everything is connected
The Matrix Revolutions,2003,Everything that has a beginning has an end
The Matrix Reloaded,2003,Free your mind
The Matrix,1999>Welcome to the Real World
Speed Racer,2008,Speed has no limits
Cloud Atlas,2012,Everything is connected
The Matrix Revolutions,2003,Everything that has a beginning has an end
The Matrix Reloaded,2003,Free your mind
The Matrix,1999>Welcome to the Real World
Ninja Assassin,2009,Prepare to enter a secret world of assassins
V for Vendetta,2006,Freedom! Forever!
Speed Racer,2008,Speed has no limits
V for Vendetta,2006,Freedom! Forever!
Speed Racer,2008,Speed has no limits
Cloud Atlas,2012,Everything is connected
The Matrix Revolutions,2003,Everything that has a beginning has an end
The Matrix Reloaded,2003,Free your mind
The Matrix,1999>Welcome to the Real World
Ninja Assassin,2009,Prepare to enter a secret world of assassins
V for Vendetta,2006,Freedom! Forever!
Speed Racer,2008,Speed has no limits
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!
The Matrix Revolutions,2003,Everything that has a beginning has an end
The Matrix Reloaded,2003,Free your mind
The Matrix,1999>Welcome to the Real World
The Matrix,1999>Welcome to the Real World
That Thing You Do,1996,In every life there comes a time when that thing you dream becomes that thing you do
The Devil's Advocate,1997,Evil has its winning ways
The Devil's Advocate,1997,Evil has its winning ways
The Devil's Advocate,1997,Evil has its winning ways
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."
Something's Gotta Give,1975,null
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.
As Good as It Gets,1997,A comedy from the heart that goes for the throat.
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."
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."
```



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."

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."

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.

Jerry Maguire,2000,The rest of his life begins now.

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."

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."

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

V 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."

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."

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."

When Harry Met Sally,1998,At odds in life... in love on-line.

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."

Top Gun,1986,"I feel the need, the need for speed."

Top Gun,1986,"I feel the need, the need for speed."

Top Gun,1986,"I feel the need, the need for speed."

Top Gun,1986,"I feel the need, the need for speed."

When Harry Met Sally,1998,At odds in life... in love on-line.

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.

Top Gun,1986,"I feel the need, the need for speed."

Top Gun,1986,"I feel the need, the need for speed."

Top Gun,1986,"I feel the need, the need for speed."

Jerry Maguire,2000,The rest of his life begins now.

Jerry Maguire,2000,The rest of his life begins now.

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.

Jerry Maguire,2000,The rest of his life begins now.

The Green Mile,1999,Walk a mile you'll never forget.

Jerry Maguire,2000,The rest of his life begins now.

Jerry Maguire,2000,The rest of his life begins now.

Jerry Maguire,2000,The rest of his life begins now.

Jerry Maguire,2000,The rest of his life begins now.

Jerry Maguire,2000,The rest of his life begins now.

Jerry Maguire,2000,The rest of his life begins now.

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."

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.

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  
Charlie Wilson's War,2007,A stiff drink. A little mascara. A lot of nerve. Who said they couldn't bring down the Soviet empire.  
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?"  
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 Replacements,2000,"Pain heals, Chicks dig scars... Glory lasts forever"  
RescueDawn,2006,Based on the extraordinary true story of one man's fight for freedom  
Twister,1996,Don't Breathe. Don't Look Back.  
RescueDawn,2006,Based on the extraordinary true story of one man's fight for freedom  
Charlie Wilson's War,2007,A stiff drink. A little mascara. A lot of nerve. Who said they couldn't bring down the Soviet empire.  
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  
 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.  
 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."  
 Charlie Wilson's War,2007,A stiff drink. A little mascara. A lot of nerve. Who said they couldn't bring down the Soviet empire.  
 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.  
 Charlie Wilson's War,2007,A stiff drink. A little mascara. A lot of nerve. Who said they couldn't bring down the Soviet empire.  
 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.  
 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:

title,roles,name,born  
 Something's Gotta Give,Julian Mercer,Keanu Reeves,1964  
 Johnny Mnemonic,Johnny Mnemonic,Keanu Reeves,1964  
 The Replacements,Shane Falco,Keanu Reeves,1964  
 The Devil's Advocate,Kevin Lomax,Keanu Reeves,1964  
 The Matrix Revolutions,Neo,Keanu Reeves,1964  
 The Matrix Reloaded,Neo,Keanu Reeves,1964  
 The Matrix,Neo,Keanu Reeves,1964  
 The Matrix Revolutions,Trinity,Carrie-Anne Moss,1967  
 The Matrix Reloaded,Trinity,Carrie-Anne Moss,1967  
 The Matrix,Trinity,Carrie-Anne Moss,1967  
 The Matrix Revolutions,Morpheus,Laurence Fishburne,1961  
 The Matrix Reloaded,Morpheus,Laurence Fishburne,1961  
 The Matrix,Morpheus,Laurence Fishburne,1961  
 V for Vendetta,V,Hugo Weaving,1960  
 Cloud Atlas,Bill Smoke;Haskell Moore;Tadeusz Kesselring;Nurse Noakes;Boardman Mephi;Old Georgie,Hugo Weaving,1960  
 The Matrix Revolutions,Agent Smith,Hugo Weaving,1960  
 The Matrix Reloaded,Agent Smith,Hugo Weaving,1960  
 The Matrix,Agent Smith,Hugo Weaving,1960  
 The Matrix,Emil,Emil Eifrem,1978  
 That Thing You Do,Tina,Charlize Theron,1975  
 The Devil's Advocate,Mary Ann Lomax,Charlize Theron,1975  
 The Devil's Advocate,John Milton,Al Pacino,1940  
 Jerry Maguire,Jerry Maguire,Tom Cruise,1962  
 Top Gun,Maverick,Tom Cruise,1962  
 A Few Good Men,Lt. Daniel Kaffee,Tom Cruise,1962  
 Something's Gotta Give,Harry Sanborn,Jack Nicholson,1937  
 One Flew Over the Cuckoo's Nest,Randle McMurphy,Jack Nicholson,1937  
 Hoffa,Hoffa,Jack Nicholson,1937  
 As Good as It Gets,Melvin Udall,Jack Nicholson,1937  
 A Few Good Men,Col. Nathan R. Jessup,Jack Nicholson,1937  
 A Few Good Men,Lt. Cdr. JoAnne Galloway,Demi Moore,1962  
 Apollo 13,Jack Swigert,Kevin Bacon,1958  
 Frost/Nixon,Jack Brennan,Kevin Bacon,1958  
 A Few Good Men,Capt. Jack Ross,Kevin Bacon,1958  
 Stand By Me,Ace Merrill,Kiefer Sutherland,1966  
 A Few Good Men,Lt. Jonathan Kendrick,Kiefer Sutherland,1966  
 A Few Good Men,Cpl. Jeffrey Barnes,Noah Wyle,1971  
 What Dreams May Come,Albert Lewis,Cuba Gooding Jr.,1968  
 As Good as It Gets,Frank Sachs,Cuba Gooding Jr.,1968  
 Jerry Maguire,Rod Tidwell,Cuba Gooding Jr.,1968  
 A Few Good Men,Cpl. Carl Hammaker,Cuba Gooding Jr.,1968  
 A Few Good Men,Lt. Sam Weinberg,Kevin Pollak,1957  
 Hoffa,Frank Fitzsimmons,J.T. Walsh,1943  
 A Few Good Men,Lt. Col. Matthew Andrew Markinson,J.T. Walsh,1943  
 A Few Good Men,Pfc. Loudon Downey,James Marshall,1967  
 A Few Good Men,Dr. Stone,Christopher Guest,1948  
 A Few Good Men,Man in Bar,Aaron Sorkin,1961  
 Top Gun,Charlie,Kelly McGillis,1957  
 Top Gun,Iceman,Val Kilmer,1959  
 Top Gun,Goose,Anthony Edwards,1962  
 Top Gun,Viper,Tom Skerritt,1933  
 When Harry Met Sally,Sally Albright,Meg Ryan,1961  
 Joe Versus the Volcano,DeDe;Angelica Graynamore;Patricia Graynamore,Meg Ryan,1961  
 Sleepless in Seattle,Annie Reed,Meg Ryan,1961  
 You've Got Mail,Kathleen Kelly,Meg Ryan,1961  
 Top Gun,Carole,Meg Ryan,1961  
 Jerry Maguire,Dorothy Boyd,Renee Zellweger,1969  
 Jerry Maguire,Avery Bishop,Kelly Preston,1962  
 Stand By Me,Vern Tessio,Jerry O'Connell,1974  
 Jerry Maguire,Frank Cushman,Jerry O'Connell,1974  
 Jerry Maguire,Bob Sugar,Jay Mohr,1970  
 The Green Mile,Jan Edgecomb,Bonnie Hunt,1961  
 Jerry Maguire,Laurel Boyd,Bonnie Hunt,1961  
 Jerry Maguire,Marcee Tidwell,Regina King,1971  
 Jerry Maguire,Ray Boyd,Jonathan Lipnicki,1990  
 Stand By Me,Chris Chambers,River Phoenix,1970  
 Stand By Me,Teddy Duchamp,Corey Feldman,1971  
 Stand By Me,Gordie Lachance,Wil Wheaton,1972  
 Stand By Me,Denny Lachance,John Cusack,1966  
 RescueDawn,Admiral,Marshall Bell,1942  
 Stand By Me,Mr. Lachance,Marshall Bell,1942  
 Cast Away,Kelly Frears,Helen Hunt,1963

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  
 RescueDawn,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  
 Charlie Wilson's War,Rep. Charlie Wilson, 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, Sam Baldwin, Tom Hanks,1956  
 You've Got Mail,Joe Fox, 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  
 RescueDawn,Dieter Dengler,Christian Bale,1974  
 Twister,Eddie,Zach Grenier,1954  
 RescueDawn,Squad Leader,Zach Grenier,1954  
 Unforgiven,English Bob,Richard Harris,1930  
 Unforgiven,Bill Munny,Clint Eastwood,1930  
 Johnny Mnemonic,Takahashi,Takeshi Kitano,1947  
 Johnny Mnemonic,Jane,Dina Meyer,1968  
 Johnny Mnemonic,J-Bone,Ice-T,1958  
 Cloud Atlas,Luisa Rey;Jocasta Ayrs;Ovid;Meronym,Halle Berry,1966  
 Cloud Atlas,Vyvyan Ayrs;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 Sutler,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

Frost/Nixon,"James Reston, Jr.",Sam Rockwell,1968  
The Green Mile,'Wild Bill' Wharton,Sam Rockwell,1968  
Apollo 13,Ken Mattingly,Gary Sinise,1955  
The Green Mile,Burt Hammersmith,Gary Sinise,1955  
The Green Mile,Melinda Moores,Patricia Clarkson,1959  
Frost/Nixon,Richard Nixon,Frank Langella,1938  
Frost/Nixon,David Frost,Michael Sheen,1969  
Bicentennial Man,Rupert Burns,Oliver Platt,1960  
Frost/Nixon,Bob Zelnick,Oliver Platt,1960  
One Flew Over the Cuckoo's Nest,Martini,Danny DeVito,1944  
Hoffa,Robert 'Bobby' Ciaro,Danny DeVito,1944  
Hoffa,Peter 'Pete' Connelly,John C. Reilly,1965  
Apollo 13,Gene Kranz,Ed Harris,1950  
A League of Their Own,Bob Hinson,Bill Paxton,1955  
Twister,Bill Harding,Bill Paxton,1955  
Apollo 13,Fred Haise,Bill Paxton,1955  
Charlie Wilson's War,Gust Avrakotos,Philip Seymour Hoffman,1967  
Twister,Dustin 'Dusty' Davis,Philip Seymour Hoffman,1967  
Something's Gotta Give,Erica Barry,Diane Keaton,1946  
Charlie Wilson's War,Joanne Herring,Julia Roberts,1967  
A League of Their Own,'All the Way' Mae Mordabito,Madonna,1954  
A League of Their Own,Dottie Hinson,Geena Davis,1956  
A League of Their Own,Kit Keller,Lori Petty,1963

## Directors

The `directors.csv` file contains two columns `title`, `name`, and `born`.

The content of the `directors.csv` file:

```

title,name,born
Speed Racer,Andy Wachowski,1967
Cloud Atlas,Andy Wachowski,1967
The Matrix Revolutions,Andy Wachowski,1967
The Matrix Reloaded,Andy Wachowski,1967
The Matrix,Andy Wachowski,1967
Speed Racer,Lana Wachowski,1965
Cloud Atlas,Lana Wachowski,1965
The Matrix Revolutions,Lana Wachowski,1965
The Matrix Reloaded,Lana Wachowski,1965
The Matrix,Lana Wachowski,1965
The Devil's Advocate,Taylor Hackford,1944
Ninja Assassin,James Marshall,1967
V for Vendetta,James Marshall,1967
When Harry Met Sally,Rob Reiner,1947
Stand By Me,Rob Reiner,1947
A Few Good Men,Rob Reiner,1947
Top Gun,Tony Scott,1944
Jerry Maguire,Cameron Crowe,1957
As Good as It Gets,James L. Brooks,1940
RescueDawn,Werner Herzog,1942
What Dreams May Come,Vincent Ward,1956
Snow Falling on Cedars,Scott Hicks,1953
That Thing You Do,Tom Hanks,1956
Sleepless in Seattle,Nora Ephron,1941
You've Got Mail,Nora Ephron,1941
Joe Versus the Volcano,John Patrick Stanley,1950
The Replacements,Howard Deutch,1950
Charlie Wilson's War,Mike Nichols,1931
The Birdcage,Mike Nichols,1931
Unforgiven,Clint Eastwood,1930
Johnny Mnemonic,Robert Longo,1953
Cloud Atlas,Tom Tykwer,1965
Apollo 13,Ron Howard,1954
Frost/Nixon,Ron Howard,1954
The Da Vinci Code,Ron Howard,1954
The Green Mile,Frank Darabont,1959
Hoffa,Danny DeVito,1944
Twister,Jan de Bont,1943
The Polar Express,Robert Zemeckis,1951
Cast Away,Robert Zemeckis,1951
One Flew Over the Cuckoo's Nest,Milos Forman,1932
Something's Gotta Give,Nancy Meyers,1949
Bicentennial Man,Chris Columbus,1958
A League of Their Own,Penny Marshall,1943

```

## 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](#).

## 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

## 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 [Cypher query options](#) but in this case you are going to prefix our query with **PROFILE**:

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



```

+-----+
| p |
+-----+
| (:Person {name: "Tom Hanks", born: 1956}) |
+-----+

+-----+
| Plan | Statement | Version | Planner | Runtime | Time | DbHits | Rows | Memory (Bytes) |
+-----+
| "PROFILE" | "READ_ONLY" | "CYPHER 4.3" | "COST" | "PIPELINED" | 26 | 406 | 1 | 136 |
+-----+

+-----+
+-----+
| Operator | Details | Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page
Cache Hits/Misses | Time (ms) | Other |
+-----+
+-----+
| +ProduceResults@neo4j | p | 8 | 1 | 3 | |
| | Fused in Pipeline 0 |
| |
+-----+
| +Filter@neo4j | p.name = $autostring_0 | 8 | 1 | 239 | |
| | Fused in Pipeline 0 |
| |
+-----+
| +AllNodesScan@neo4j | p | 163 | 163 | 164 | 72 |
4/0 | 1.705 | Fused in Pipeline 0 |
+-----+
+-----+

1 row

```

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 you notice is that the value in the **Rows** column seems high given there is only one node with the name property 'Tom Hanks' in the database. If you look across to the **Operator** column, you will see that **AllNodesScan** has been used which means that the query planner scanned through all the nodes in the database.

The **Filter** operator which will check the **name** property on each of the nodes passed through by **AllNodesScan**.

This seems like an inefficient way of finding 'Tom Hanks' given that you are looking at many nodes that are not even people and therefore are not what you are looking for.

The solution to this problem is that whenever you are looking for a node you should specify a label to help the query planner narrow down the search space.

For this query you need to add a **Person** label.

```

MATCH (p:Person {name: 'Tom Hanks'})
RETURN p

```

This query will be faster than the first one, but as the number of people in your database increase you may notice that the query slows down.

Again you can profile the query to work out why:

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

```
+-----+
| p |
+-----+
| (:Person {name: "Tom Hanks", born: 1956}) |
+-----+

+-----+-----+-----+-----+-----+-----+-----+-----+
| Plan      | Statement | Version      | Planner | Runtime      | Time | DbHits | Rows | Memory (Bytes) |
+-----+-----+-----+-----+-----+-----+-----+-----+
| "PROFILE" | "READ_ONLY" | "CYPHER 4.3" | "COST"  | "PIPELINED" | 33   | 379   | 1   | 136             |
+-----+-----+-----+-----+-----+-----+-----+-----+

+-----+-----+-----+-----+-----+-----+-----+-----+
| Operator          | Details              | Estimated Rows | Rows | DB Hits | Memory (Bytes) |
| Page Cache Hits/Misses | Time (ms) | Other          |      |         |                |
+-----+-----+-----+-----+-----+-----+-----+-----+
| +ProduceResults@neo4j | p                  | 6              | 1   | 3       |                | |
| | Fused in Pipeline 0 |                    |                |     |         |                |
| |                    |                    |                |     |         |                |
+-----+-----+-----+-----+-----+-----+-----+-----+
| +Filter@neo4j      | p.name = $autostring_0 | 6              | 1   | 250     |                | |
| | Fused in Pipeline 0 |                    |                |     |         |                |
| |                    |                    |                |     |         |                |
+-----+-----+-----+-----+-----+-----+-----+-----+
| +NodeByLabelScan@neo4j | p:Person          | 125            | 125 | 126     |                |
| 4/0 | 0.901 | Fused in Pipeline 0 |                    |                |     |         |                |
+-----+-----+-----+-----+-----+-----+-----+-----+

1 row
```

This time the **Rows** value on the last row has reduced so you are not scanning some nodes that you were before which is a good start. The **NodeByLabelScan** operator indicates that you achieved this by first doing a linear scan of all the **Person** nodes in the database.

Once you have done that, you can 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 you are going to be looking up people by name frequently then you will see better performance if you 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 you run the query again it will run more quickly:

```
MATCH (p:Person {name: 'Tom Hanks'})
RETURN p
```

A profile for the query to see why that is:

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

```
+-----+
| p |
+-----+
| (:Person {name: "Tom Hanks", born: 1956}) |
+-----+

+-----+
| Plan | Statement | Version | Planner | Runtime | Time | DbHits | Rows | Memory (Bytes) |
+-----+
| "PROFILE" | "READ_ONLY" | "CYPHER 4.3" | "COST" | "PIPELINED" | 17 | 5 | 1 | 136 |
+-----+

+-----+
+-----+
| Operator | Details | Estimated Rows | Rows | DB Hits |
Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Other |
+-----+
| +ProduceResults@neo4j | p | | 1 | 1 | 3 |
| | | Fused in Pipeline 0 |
| | +-----+
+-----+
| +NodeIndexSeek@neo4j | p:Person(name) WHERE name = $autostring_0 | 1 | 1 | 2 |
72 | 2/1 | 0.494 | Fused in Pipeline 0 |
+-----+
+-----+

1 row
```

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.

## 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 4.4.28, 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](#).

## 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:

`movies.csv`

```
title,released,tagline
Something's Gotta Give,1975,null
Johnny Mnemonic,1995,The hottest data on earth. In the coolest head in town
The Replacements,2000,"Pain heals, Chicks dig scars... Glory lasts forever"
The Devil's Advocate,1997,Evil has its winning ways
The Matrix Revolutions,2003,Everything that has a beginning has an end
The Matrix Reloaded,2003,Free your mind
The Matrix,1999>Welcome to the Real World
The Matrix Revolutions,2003,Everything that has a beginning has an end
The Matrix Reloaded,2003,Free your mind
The Matrix,1999>Welcome to the Real World
The Matrix Revolutions,2003,Everything that has a beginning has an end
The Matrix Reloaded,2003,Free your mind
The Matrix,1999>Welcome to the Real World
V for Vendetta,2006,Freedom! Forever!
Cloud Atlas,2012,Everything is connected
The Matrix Revolutions,2003,Everything that has a beginning has an end
The Matrix Reloaded,2003,Free your mind
The Matrix,1999>Welcome to the Real World
Speed Racer,2008,Speed has no limits
Cloud Atlas,2012,Everything is connected
The Matrix Revolutions,2003,Everything that has a beginning has an end
The Matrix Reloaded,2003,Free your mind
The Matrix,1999>Welcome to the Real World
Ninja Assassin,2009,Prepare to enter a secret world of assassins
V for Vendetta,2006,Freedom! Forever!
Speed Racer,2008,Speed has no limits
V for Vendetta,2006,Freedom! Forever!
Speed Racer,2008,Speed has no limits
Cloud Atlas,2012,Everything is connected
The Matrix Revolutions,2003,Everything that has a beginning has an end
The Matrix Reloaded,2003,Free your mind
The Matrix,1999>Welcome to the Real World
Ninja Assassin,2009,Prepare to enter a secret world of assassins
V for Vendetta,2006,Freedom! Forever!
Speed Racer,2008,Speed has no limits
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!
The Matrix Revolutions,2003,Everything that has a beginning has an end
The Matrix Reloaded,2003,Free your mind
The Matrix,1999>Welcome to the Real World
The Matrix,1999>Welcome to the Real World
That Thing You Do,1996,In every life there comes a time when that thing you dream becomes that thing you do
The Devil's Advocate,1997,Evil has its winning ways
```

The Devil's Advocate,1997,Evil has its winning ways  
The Devil's Advocate,1997,Evil has its winning ways  
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."  
Something's Gotta Give,1975,null  
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.  
As Good as It Gets,1997,A comedy from the heart that goes for the throat.  
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."  
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."  
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."  
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."  
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.  
Jerry Maguire,2000,The rest of his life begins now.  
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."  
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."  
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  
V 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."  
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."  
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."  
When Harry Met Sally,1998,At odds in life... in love on-line.  
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."  
Top Gun,1986,"I feel the need, the need for speed."  
Top Gun,1986,"I feel the need, the need for speed."  
Top Gun,1986,"I feel the need, the need for speed."  
Top Gun,1986,"I feel the need, the need for speed."  
When Harry Met Sally,1998,At odds in life... in love on-line.  
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.  
Top Gun,1986,"I feel the need, the need for speed."  
Top Gun,1986,"I feel the need, the need for speed."  
Top Gun,1986,"I feel the need, the need for speed."  
Jerry Maguire,2000,The rest of his life begins now.  
Jerry Maguire,2000,The rest of his life begins now.  
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.  
Jerry Maguire,2000,The rest of his life begins now.  
The Green Mile,1999,Walk a mile you'll never forget.  
Jerry Maguire,2000,The rest of his life begins now.  
Jerry Maguire,2000,The rest of his life begins now.  
Jerry Maguire,2000,The rest of his life begins now.  
Jerry Maguire,2000,The rest of his life begins now.  
Jerry Maguire,2000,The rest of his life begins now.  
Jerry Maguire,2000,The rest of his life begins now.  
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."  
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.  
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  
Charlie Wilson's War,2007,A stiff drink. A little mascara. A lot of nerve. Who said they couldn't bring  
down the Soviet empire.  
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.  
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 Replacements,2000,"Pain heals, Chicks dig scars... Glory lasts forever"  
RescueDawn,2006,Based on the extraordinary true story of one man's fight for freedom  
Twister,1996,Don't Breathe. Don't Look Back.  
RescueDawn,2006,Based on the extraordinary true story of one man's fight for freedom  
Charlie Wilson's War,2007,A stiff drink. A little mascara. A lot of nerve. Who said they couldn't bring down the Soviet empire.  
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  
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.  
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.  
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."  
Charlie Wilson's War,2007,A stiff drink. A little mascara. A lot of nerve. Who said they couldn't bring down the Soviet empire.  
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.  
Charlie Wilson's War,2007,A stiff drink. A little mascara. A lot of nerve. Who said they couldn't bring down the Soviet empire.  
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.  
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:

### actors.csv

```
title,roles,name,born
Something's Gotta Give,Julian Mercer,Keanu Reeves,1964
Johnny Mnemonic,Johnny Mnemonic,Keanu Reeves,1964
The Replacements,Shane Falco,Keanu Reeves,1964
The Devil's Advocate,Kevin Lomax,Keanu Reeves,1964
The Matrix Revolutions,Neo,Keanu Reeves,1964
The Matrix Reloaded,Neo,Keanu Reeves,1964
The Matrix,Neo,Keanu Reeves,1964
The Matrix Revolutions,Trinity,Carrie-Anne Moss,1967
The Matrix Reloaded,Trinity,Carrie-Anne Moss,1967
The Matrix,Trinity,Carrie-Anne Moss,1967
The Matrix Revolutions,Morpheus,Laurence Fishburne,1961
The Matrix Reloaded,Morpheus,Laurence Fishburne,1961
The Matrix,Morpheus,Laurence Fishburne,1961
V for Vendetta,V,Hugo Weaving,1960
Cloud Atlas,Bill Smoke;Haskell Moore;Tadeusz Kesselring;Nurse Noakes;Boardman Mephi;Old Georgie,Hugo Weaving,1960
The Matrix Revolutions,Agent Smith,Hugo Weaving,1960
The Matrix Reloaded,Agent Smith,Hugo Weaving,1960
The Matrix,Agent Smith,Hugo Weaving,1960
The Matrix,Emil,Emil Eifrem,1978
That Thing You Do,Tina,Charlize Theron,1975
The Devil's Advocate,Mary Ann Lomax,Charlize Theron,1975
The Devil's Advocate,John Milton,Al Pacino,1940
Jerry Maguire,Jerry Maguire,Tom Cruise,1962
Top Gun,Maverick,Tom Cruise,1962
A Few Good Men,Lt. Daniel Kaffee,Tom Cruise,1962
Something's Gotta Give,Harry Sanborn,Jack Nicholson,1937
One Flew Over the Cuckoo's Nest,Randle McMurphy,Jack Nicholson,1937
Hoffa,Hoffa,Jack Nicholson,1937
As Good as It Gets,Melvin Udall,Jack Nicholson,1937
A Few Good Men,Col. Nathan R. Jessup,Jack Nicholson,1937
A Few Good Men,Lt. Cdr. JoAnne Galloway,Demi Moore,1962
Apollo 13,Jack Swigert,Kevin Bacon,1958
Frost/Nixon,Jack Brennan,Kevin Bacon,1958
A Few Good Men,Capt. Jack Ross,Kevin Bacon,1958
Stand By Me,Ace Merrill,Kiefer Sutherland,1966
A Few Good Men,Lt. Jonathan Kendrick,Kiefer Sutherland,1966
A Few Good Men,Cpl. Jeffrey Barnes,Noah Wyle,1971
What Dreams May Come,Albert Lewis,Cuba Gooding Jr.,1968
As Good as It Gets,Frank Sachs,Cuba Gooding Jr.,1968
Jerry Maguire,Rod Tidwell,Cuba Gooding Jr.,1968
A Few Good Men,Cpl. Carl Hammaker,Cuba Gooding Jr.,1968
A Few Good Men,Lt. Sam Weinberg,Kevin Pollak,1957
Hoffa,Frank Fitzsimmons,J.T. Walsh,1943
A Few Good Men,Lt. Col. Matthew Andrew Markinson,J.T. Walsh,1943
A Few Good Men,Pfc. Loudon Downey,James Marshall,1967
A Few Good Men,Dr. Stone,Christopher Guest,1948
A Few Good Men,Man in Bar,Aaron Sorkin,1961
Top Gun,Charlie,Kelly McGillis,1957
Top Gun,Iceman,Val Kilmer,1959
Top Gun,Goose,Anthony Edwards,1962
Top Gun,Viper,Tom Skerritt,1933
When Harry Met Sally,Sally Albright,Meg Ryan,1961
Joe Versus the Volcano,DeDe;Angelica Graynamore;Patricia Graynamore,Meg Ryan,1961
Sleepless in Seattle,Annie Reed,Meg Ryan,1961
You've Got Mail,Kathleen Kelly,Meg Ryan,1961
```



Top Gun, Carole, Meg Ryan, 1961  
 Jerry Maguire, Dorothy Boyd, Renee Zellweger, 1969  
 Jerry Maguire, Avery Bishop, Kelly Preston, 1962  
 Stand By Me, Vern Tessio, Jerry O'Connell, 1974  
 Jerry Maguire, Frank Cushman, Jerry O'Connell, 1974  
 Jerry Maguire, Bob Sugar, Jay Mohr, 1970  
 The Green Mile, Jan Edgecomb, Bonnie Hunt, 1961  
 Jerry Maguire, Laurel Boyd, Bonnie Hunt, 1961  
 Jerry Maguire, Marcee Tidwell, Regina King, 1971  
 Jerry Maguire, Ray Boyd, Jonathan Lipnicki, 1990  
 Stand By Me, Chris Chambers, River Phoenix, 1970  
 Stand By Me, Teddy Duchamp, Corey Feldman, 1971  
 Stand By Me, Gordie Lachance, Wil Wheaton, 1972  
 Stand By Me, Denny Lachance, John Cusack, 1966  
 Rescue Dawn, Admiral, Marshall Bell, 1942  
 Stand By Me, Mr. Lachance, Marshall Bell, 1942  
 Cast Away, Kelly Frears, Helen Hunt, 1963  
 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  
 Charlie Wilson's War, Rep. Charlie Wilson, 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, Sam Baldwin, Tom Hanks, 1956  
 You've Got Mail, Joe Fox, 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, 1930  
 Unforgiven, Bill Munny, Clint Eastwood, 1930  
 Johnny Mnemonic, Takahashi, Takeshi Kitano, 1947  
 Johnny Mnemonic, Jane, Dina Meyer, 1968  
 Johnny Mnemonic, J-Bone, Ice-T, 1958  
 Cloud Atlas, Luisa Rey; Jocasta Ayrs; Ovid; Meronym, Halle Berry, 1966  
 Cloud Atlas, Vyvyan Ayrs; 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 Sutler,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  
 Frost/Nixon,"James Reston, Jr.",Sam Rockwell,1968  
 The Green Mile,'Wild Bill' Wharton,Sam Rockwell,1968  
 Apollo 13,Ken Mattingly,Gary Sinise,1955  
 The Green Mile,Burt Hammersmith,Gary Sinise,1955  
 The Green Mile,Melinda Moores,Patricia Clarkson,1959  
 Frost/Nixon,Richard Nixon,Frank Langella,1938  
 Frost/Nixon,David Frost,Michael Sheen,1969  
 Bicentennial Man,Rupert Burns,Oliver Platt,1960  
 Frost/Nixon,Bob Zelnick,Oliver Platt,1960  
 One Flew Over the Cuckoo's Nest,Martini,Danny DeVito,1944  
 Hoffa,Robert 'Bobby' Ciaro,Danny DeVito,1944  
 Hoffa,Peter 'Pete' Connelly,John C. Reilly,1965  
 Apollo 13,Gene Kranz,Ed Harris,1950  
 A League of Their Own,Bob Hinson,Bill Paxton,1955  
 Twister,Bill Harding,Bill Paxton,1955  
 Apollo 13,Fred Haise,Bill Paxton,1955  
 Charlie Wilson's War,Gust Avrakotos,Philip Seymour Hoffman,1967  
 Twister,Dustin 'Dusty' Davis,Philip Seymour Hoffman,1967  
 Something's Gotta Give,Erica Barry,Diane Keaton,1946  
 Charlie Wilson's War,Joanne Herring,Julia Roberts,1967  
 A League of Their Own,'All the Way' Mae Mordabito,Madonna,1954  
 A League of Their Own,Dottie Hinson,Geena Davis,1956  
 A League of Their Own,Kit Keller,Lori Petty,1963

## Directors

The `directors.csv` file contains two columns `title`, `name`, and `born`.

The content of the `directors.csv` file:

```

title,name,born
Speed Racer,Andy Wachowski,1967
Cloud Atlas,Andy Wachowski,1967
The Matrix Revolutions,Andy Wachowski,1967
The Matrix Reloaded,Andy Wachowski,1967
The Matrix,Andy Wachowski,1967
Speed Racer,Lana Wachowski,1965
Cloud Atlas,Lana Wachowski,1965
The Matrix Revolutions,Lana Wachowski,1965
The Matrix Reloaded,Lana Wachowski,1965
The Matrix,Lana Wachowski,1965
The Devil's Advocate,Taylor Hackford,1944
Ninja Assassin,James Marshall,1967
V for Vendetta,James Marshall,1967
When Harry Met Sally,Rob Reiner,1947
Stand By Me,Rob Reiner,1947
A Few Good Men,Rob Reiner,1947
Top Gun,Tony Scott,1944
Jerry Maguire,Cameron Crowe,1957
As Good as It Gets,James L. Brooks,1940
RescueDawn,Werner Herzog,1942
What Dreams May Come,Vincent Ward,1956
Snow Falling on Cedars,Scott Hicks,1953
That Thing You Do,Tom Hanks,1956
Sleepless in Seattle,Nora Ephron,1941
You've Got Mail,Nora Ephron,1941
Joe Versus the Volcano,John Patrick Stanley,1950
The Replacements,Howard Deutch,1950
Charlie Wilson's War,Mike Nichols,1931
The Birdcage,Mike Nichols,1931
Unforgiven,Clint Eastwood,1930
Johnny Mnemonic,Robert Longo,1953
Cloud Atlas,Tom Tykwer,1965
Apollo 13,Ron Howard,1954
Frost/Nixon,Ron Howard,1954
The Da Vinci Code,Ron Howard,1954
The Green Mile,Frank Darabont,1959
Hoffa,Danny DeVito,1944
Twister,Jan de Bont,1943
The Polar Express,Robert Zemeckis,1951
Cast Away,Robert Zemeckis,1951
One Flew Over the Cuckoo's Nest,Milos Forman,1932
Something's Gotta Give,Nancy Meyers,1949
Bicentennial Man,Chris Columbus,1958
A League of Their Own,Penny Marshall,1943

```

## 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](#).

## 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
```

## 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
```

```

+-----+
| name          | count |
+-----+
| "Tom Cruise"  | 3     |
| "Tom Hanks"   | 12    |
| "Tom Skerritt"| 1     |
+-----+
3 rows

```

The query request 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.

```

PROFILE
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WHERE p.name STARTS WITH 'Tom'
RETURN
  p.name AS name,
  count(m) AS count

```

```

+-----+
| name          | count |
+-----+
| "Tom Cruise"  | 3     |
| "Tom Hanks"   | 12    |
| "Tom Skerritt"| 1     |
+-----+

+-----+
| Plan          | Statement | Version      | Planner | Runtime      | Time | DbHits | Rows | Memory (Bytes) |
+-----+
| "PROFILE"    | "READ_ONLY" | "CYPHER 4.3" | "COST"  | "PIPELINED" | 2    | 43    | 3    | 1768           |
+-----+

+-----+
+-----+
+-----+
| Operator              | Details |
+-----+
Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Ordered by | Other |
+-----+
+-----+
| +ProduceResults@neo4j | name, count |
1 | 3 | 0 | | | 0/0 | 0.049 | name ASC | In Pipeline 1 |
| |
+-----+
+-----+
| +OrderedAggregation@neo4j | cache[p.name] AS name, count(m) AS count |
1 | 3 | 0 | 1688 | | 0/0 | 0.188 | name ASC | In Pipeline 1 |
| |
+-----+
+-----+
| +Filter@neo4j | m:Movie |
1 | 16 | 16 | | | | p.name ASC | Fused in Pipeline |
0 |
| |
+-----+
+-----+
| +Expand(All)@neo4j | (p)-[anon_16:ACTED_IN]->(m) |
1 | 16 | 22 | | | | p.name ASC | Fused in Pipeline |
0 |
| |
+-----+
+-----+
| +NodeIndexSeekByRange@neo4j | p:Person(name) WHERE name STARTS WITH $autostring_0, cache[p.name] |
1 | 4 | 5 | 72 | | 4/0 | 0.340 | p.name ASC | Fused in Pipeline |
0 |
+-----+
+-----+
+-----+
3 rows

```

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.

```

PROFILE
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
RETURN
  p.name AS name,
  count(m) AS count

```

```

+-----+
| name          | count |
+-----+
| "Diane Keaton" | 1     |
+-----+

```

"Jack Nicholson"	5
"Keanu Reeves"	7
"Ice-T"	1
"Takeshi Kitano"	1
"Dina Meyer"	1
"Brooke Langton"	1
"Gene Hackman"	3
"Orlando Jones"	1
"Al Pacino"	1
"Charlize Theron"	2
"Hugo Weaving"	5
"Laurence Fishburne"	3
"Carrie-Anne Moss"	3
"Emil Eifrem"	1
"John Hurt"	1
"Stephen Rea"	1
"Natalie Portman"	1
"Ben Miles"	3
"Jim Broadbent"	1
"Tom Hanks"	12
"Halle Berry"	1
"John Goodman"	1
"Susan Sarandon"	1
"Christina Ricci"	1
"Rain"	2
"Emile Hirsch"	1
"Matthew Fox"	1
"Rick Yune"	2
"Naomie Harris"	1
"Liv Tyler"	1
"Kelly Preston"	1
"Bonnie Hunt"	2
"Jerry O'Connell"	2
"Renee Zellweger"	1
"Jay Mohr"	1
"Jonathan Lipnicki"	1
"Cuba Gooding Jr."	4
"Regina King"	1
"Tom Cruise"	3
"Kelly McGillis"	1
"Anthony Edwards"	1
"Tom Skerritt"	1
"Meg Ryan"	5
"Val Kilmer"	1
"Kiefer Sutherland"	2
"Kevin Bacon"	3
"Aaron Sorkin"	1
"Christopher Guest"	1
"Noah Wyle"	1
"James Marshall"	1
"Kevin Pollak"	1
"J.T. Walsh"	2
"Demi Moore"	1
"Danny DeVito"	2
"John C. Reilly"	1
"Helen Hunt"	3
"Greg Kinnear"	2
"Ed Harris"	1
"Bill Paxton"	3
"Gary Sinise"	2
"Oliver Platt"	2
"Frank Langella"	1
"Michael Sheen"	1
"Sam Rockwell"	2
"John Cusack"	1
"Wil Wheaton"	1
"Corey Feldman"	1
"River Phoenix"	1
"Marshall Bell"	2
"Max von Sydow"	2
"Annabella Sciorra"	1
"Werner Herzog"	1
"Robin Williams"	3
"Billy Crystal"	1
"Carrie Fisher"	1
"Bruno Kirby"	1
"Nathan Lane"	2

```

| "Rita Wilson" | 1 |
| "Rosie O'Donnell" | 2 |
| "Bill Pullman" | 1 |
| "Victor Garber" | 1 |
| "Steve Zahn" | 2 |
| "Dave Chappelle" | 1 |
| "Parker Posey" | 1 |
| "James Cromwell" | 2 |
| "Patricia Clarkson" | 1 |
| "Michael Clarke Duncan" | 1 |
| "David Morse" | 1 |
| "Zach Grenier" | 2 |
| "Christian Bale" | 1 |
| "Philip Seymour Hoffman" | 2 |
| "Ethan Hawke" | 1 |
| "Geena Davis" | 1 |
| "Madonna" | 1 |
| "Lori Petty" | 1 |
| "Julia Roberts" | 1 |
| "Ian McKellen" | 1 |
| "Paul Bettany" | 1 |
| "Audrey Tautou" | 1 |
| "Clint Eastwood" | 1 |
| "Richard Harris" | 1 |
+-----+

```

```

+-----+
| Plan | Statement | Version | Planner | Runtime | Time | DbHits | Rows | Memory (Bytes) |
+-----+
| "PROFILE" | "READ_ONLY" | "CYPHER 4.3" | "COST" | "PIPELINED" | 70 | 809 | 102 | 17376 |
+-----+

```

```

+-----+
| Operator | Details | Estimated Rows | Rows | DB Hits | Memory
(Bytes) | Page Cache Hits/Misses | Time (ms) | Other |
+-----+
| +ProduceResults@neo4j | name, count | 13 | 102 | 0 |
| | 0/0 | 0.536 | In Pipeline 1 |
| |
+-----+
| +EagerAggregation@neo4j | p.name AS name, count(m) AS count | 13 | 102 | 344 |
17296 | | Fused in Pipeline 0 |
| |
+-----+
| +Filter@neo4j | p:Person | 172 | 172 | 172 |
| | | Fused in Pipeline 0 |
| |
+-----+
| +Expand(All)@neo4j | (m)<-[anon_16:ACTED_IN]-(p) | 172 | 172 | 254 |
| | | Fused in Pipeline 0 |
| |
+-----+
| +NodeByLabelScan@neo4j | m:Movie | 38 | 38 | 39 |
72 | 5/0 | 12.818 | Fused in Pipeline 0 |
+-----+

```

102 rows

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'`)



- 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:

```
PROFILE
MATCH (p:Person)
RETURN count(DISTINCT p.name) AS numberOfNames
```

```
+-----+
| numberOfNames |
+-----+
| 125           |
+-----+

+-----+-----+-----+-----+-----+-----+-----+-----+
| Plan          | Statement | Version   | Planner | Runtime   | Time | DbHits | Rows | Memory (Bytes) |
+-----+-----+-----+-----+-----+-----+-----+-----+
| "PROFILE"    | "READ_ONLY" | "CYPHER 4.3" | "COST"  | "PIPELINED" | 45   | 126   | 1   | 9952            |
+-----+-----+-----+-----+-----+-----+-----+-----+

+-----+-----+-----+-----+-----+-----+-----+-----+
| Operator      | Details                                     | Estimated Rows | Rows |
DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Other |
+-----+-----+-----+-----+-----+-----+-----+-----+
| +ProduceResults@neo4j | numberOfNames                               | 1 | 1 |
0 | 0 | 0/0 | 0.048 | In Pipeline 1 |
| | | | |
+-----+-----+-----+-----+-----+-----+-----+-----+
| +EagerAggregation@neo4j | count(DISTINCT cache[p.name]) AS numberOfNames | 1 | 1 |
0 | 9888 | | | Fused in Pipeline 0 |
| | | | |
+-----+-----+-----+-----+-----+-----+-----+-----+
| +NodeIndexScan@neo4j | p:Person(name) WHERE name IS NOT NULL, cache[p.name] | 125 | 125 |
126 | 72 | 1/0 | 1.569 | Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+-----+-----+-----+

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.

# Index-backed ORDER BY

Now consider the following refinement to the query:

```
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
```

```
+-----+
| name          | count |
+-----+
| "Tom Cruise"  | 3     |
| "Tom Hanks"   | 12    |
| "Tom Skerritt"| 1     |
+-----+
```

```
+-----+
| Plan          | Statement | Version      | Planner | Runtime   | Time | DbHits | Rows | Memory (Bytes) |
+-----+
| "PROFILE"    | "READ_ONLY" | "CYPHER 4.3" | "COST"  | "PIPELINED" | 48   | 43    | 3   | 1768           |
+-----+
```

```
+-----+
+-----+
+-----+
| Operator          | Details |
+-----+
Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Ordered by | Other |
+-----+
+-----+
| +ProduceResults@neo4j | name, count |
1 | 3 | 0 | | 0/0 | 0.045 | name ASC | In Pipeline 1 |
| | | | | | | |
+-----+
| +OrderedAggregation@neo4j | cache[p.name] AS name, count(m) AS count |
1 | 3 | 0 | 1688 | 0/0 | 0.173 | name ASC | In Pipeline 1 |
| | | | | | | |
+-----+
| +Filter@neo4j | m:Movie |
1 | 16 | 16 | | | | | p.name ASC | Fused in Pipeline |
0 | | | | | | |
+-----+
| +Expand(All)@neo4j | (p)-[anon_16:ACTED_IN]->(m) |
1 | 16 | 22 | | | | | p.name ASC | Fused in Pipeline |
0 | | | | | | |
+-----+
| +NodeIndexSeekByRange@neo4j | p:Person(name) WHERE name STARTS WITH $autostring_0, cache[p.name] |
1 | 4 | 5 | 72 | 4/0 | 0.459 | p.name ASC | Fused in Pipeline |
0 | | | | | | |
+-----+
+-----+
```

3 rows

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 first actor in alphabetical order:

```
PROFILE
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
RETURN min(p.name) AS name
```

```

+-----+
| name |
+-----+
| "Aaron Sorkin" |
+-----+

+-----+
| Plan | Statement | Version | Planner | Runtime | Time | DbHits | Rows | Memory (Bytes) |
+-----+
| "PROFILE" | "READ_ONLY" | "CYPHER 4.3" | "COST" | "PIPELINED" | 38 | 809 | 1 | 184 |
+-----+

+-----+
+-----+
| Operator | Details | Estimated Rows | Rows | DB Hits | Memory (Bytes) |
| Page Cache Hits/Misses | Time (ms) | Other |
+-----+
+-----+
| +ProduceResults@neo4j | name | 1 | 1 | 0 |
| | 0/0 | 0.041 | In Pipeline 1 |
| |
+-----+
| +EagerAggregation@neo4j | min(p.name) AS name | 1 | 1 | 344 | 32
| | | Fused in Pipeline 0 |
| |
+-----+
| +Filter@neo4j | p:Person | 172 | 172 | 172 |
| | | Fused in Pipeline 0 |
| |
+-----+
| +Expand(All)@neo4j | (m)<-[anon_16:ACTED_IN]-(p) | 172 | 172 | 254 |
| | | Fused in Pipeline 0 |
| |
+-----+
| +NodeByLabelScan@neo4j | m:Movie | 38 | 38 | 39 | 72
| | 5/0 | 1.636 | Fused in Pipeline 0 |
+-----+
+-----+

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'**)

- 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 4.4.28, 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`.

## 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.

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.
- `PERIODIC COMMIT` query hint. Deprecated



## 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:

Hint	Fulfilled by plans
<code>USING [BTREE   TEXT] INDEX variable:Label(property)</code>	NodeIndexScan, NodeIndexSeek
<code>USING [BTREE   TEXT] INDEX SEEK variable:Label(property)</code>	NodeIndexSeek
<code>USING [BTREE   TEXT] INDEX variable:RELATIONSHIP_TYPE(property)</code>	DirectedRelationshipIndexScan, UndirectedRelationshipIndexScan, DirectedRelationshipIndexSeek, UndirectedRelationshipIndexSeek
<code>USING [BTREE   TEXT] INDEX SEEK variable:RELATIONSHIP_TYPE(property)</code>	DirectedRelationshipIndexSeek, UndirectedRelationshipIndexSeek

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   Fused in Pipeline 0	0	0	0
+Filter	cc.formed = \$autoint_3 AND cc:Country   Fused in Pipeline 0	0	0	0
+Expand(All)	(c)-[anon_2:PART_OF]->(cc)   Fused in Pipeline 0	0	0	0
+Filter	c:City   Fused in Pipeline 0	0	0	0
+Expand(All)	(p)-[anon_1:LIVES_IN]->(c)   Fused in Pipeline 0	0	0	0
+Filter	s.born = \$autoint_0 AND s:Scientist   Fused in Pipeline 0	0	0	0
+Expand(All)	(sc)<-[anon_0:RESEARCHED]-(s)   Fused in Pipeline 0	0	0	0
+Filter	i.year = \$autoint_1 AND sc:Science   Fused in Pipeline 0	0	0	2
+Expand(All)	(p)-[i:INVENTED_BY]->(sc)   Fused in Pipeline 0	2	2	6
+NodeIndexSeek	BTREE INDEX p:Pioneer(born) WHERE born = \$autoint_2   Fused in Pipeline 0 112   4/1   0.796	2	2	3

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.

### Query

```
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
+ProduceResults	c	1	1	
+NodeIndexSeek	TEXT INDEX c:Country(name) WHERE name = \$autostring_0	1	1	

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

```

+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+
+-----+
| 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                                 | p.born = $autoint_2 AND sc:Science AND p:Pioneer |
+-----+-----+-----+-----+-----+-----+-----+
0 | 0 | 4 | | | | Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+-----+-----+
+-----+
| +DirectedRelationshipIndexSeek          | BTREE INDEX (p)-[i:INVENTED_BY(year)]->(sc) WHERE year = $autoint_1 |
+-----+-----+-----+-----+-----+-----+-----+
2 | 2 | 5 | 112 | 5/1 | 0.745 | Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+-----+-----+
+-----+

```

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

```
+-----+
+-----+
+-----+-----+-----+-----+-----+-----+
| Operator                | Details |
| Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Other |
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
| +ProduceResults        | i       |
| 1 | 1 | 0 | | | | Fused in |
Pipeline 0 |
| |
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
| +DirectedRelationshipIndexSeek | TEXT INDEX (anon_0)-[i:INVENTED_BY(location)]->(anon_1) WHERE location = $autostring_0 |
| 1 | 1 | 3 | 112 | 3/0 | 2.155 |
Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
```

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   0.000   In Pipeline 2	0	0	
+NodeHashJoin	sc 0   432     In Pipeline 2	0	0	
+Expand(All)	(s)-[anon_0:RESEARCHED]->(sc) 0     Fused in Pipeline 1	1	0	
+NodeIndexSeek	BTREE INDEX s:Scientist(born) WHERE born = \$autoint_0 0   112   0/0   0.000   Fused in Pipeline 1	1	0	
+Filter	i.year = \$autoint_1 AND sc:Science 0     Fused in Pipeline 0	0	0	
+Expand(All)	(p)-[i:INVENTED_BY]->(sc) 0     Fused in Pipeline 0	0	0	
+Filter	p.born = \$autoint_2 AND p:Pioneer 2     Fused in Pipeline 0	0	0	
+Expand(All)	(c)<-[anon_1:LIVES_IN]-(p) 3     Fused in Pipeline 0	1	1	
+Filter	c:City 1     Fused in Pipeline 0	1	1	
+Expand(All)	(cc)<-[anon_2:PART_OF]-(c) 2     Fused in Pipeline 0	1	1	
+NodeIndexSeek	BTREE INDEX cc:Country(formed) WHERE formed = \$autoint_3 2   112   7/0   1.289   Fused in Pipeline 0	1	1	

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.

## 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

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  | country                                     |
1 | 1 | 0 |          |          |          | Fused in Pipeline 2 |
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
| +Distinct        | country                                     |
1 | 1 | 0 |        224 |          |          | Fused in Pipeline 2 |
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
| +Union           |          |
2 | 1 | 0 |        1128 |          | 1/0 | 0.510 | Fused in Pipeline 2 |
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
| | +NodeIndexSeek | BTREE INDEX country:Country(formed) WHERE formed = $autoint_0 |
1 | 1 | 2 |        112 |          | 1/0 | 0.268 | In Pipeline 1 |
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
| +NodeIndexSeekByRange | BTREE INDEX country:Country(name) WHERE name STARTS WITH $autostring_1 |
1 | 0 | 1 |        112 |          | 0/1 | 0.465 | In Pipeline 0 |
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
```

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.

## 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
1	Fused in Pipeline 0		
+Expand(All)	(s)-[anon_0:RESEARCHED]->(sc)	1	1
2	Fused in Pipeline 0		
+Filter	s.born = \$autoint_0	1	1
200	Fused in Pipeline 0		
+NodeByLabelScan	s:Scientist	100	100
101	112   11/0   1.225   Fused in Pipeline 0		

Total database accesses: 308, total allocated memory: 208

## Hinting a relationship type 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 i:INVENTED_BY
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 p.born = $autoint_2 AND sc:Science AND p:Pioneer
| 0 | 0 | 204 | | | | | Fused in
Pipeline 0 |
| |
+-----+-----+-----+-----+-----+-----+
| +DirectedRelationshipTypeScan            | (p)-[i:INVENTED_BY]->(sc)
| 100 | 100 | 201 | 112 | | 9/0 | 4.182 | Fused in
Pipeline 0 |
+-----+-----+-----+-----+-----+-----+

```

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)	Page Cache
+ProduceResults	person	180	200	0		
4/0	3.343   person ASC   In Pipeline 2					
+OrderedDistinct	person	180	200	0		32
0/0	1.718   person ASC   In Pipeline 2					
+OrderedUnion	person	200	200	0		1128
0/0	1.198   person ASC   In Pipeline 2					
+NodeByLabelScan	person:Scientist	100	100	101		112
1/0	0.387   person ASC   In Pipeline 1					
+NodeByLabelScan	person:Pioneer	100	100	101		112
1/0	0.471   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.

## 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

```
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(formed)
USING JOIN ON p
RETURN *
```

## Query plan

Compiler CYPHER 4.4

Planner COST

Runtime PIPELINED

Runtime version 4.4

```

+-----+-----+-----+-----+-----+-----+-----+
| Operator      | Details                                     | Estimated Rows |
+-----+-----+-----+-----+-----+-----+-----+
| +ProduceResults | c, cc, i, p, s, sc                         |                |
0 | 0 |          | 0/0 | 0.000 | In Pipeline 2 |                |
| |          |-----+-----+-----+-----+-----+
| +NodeHashJoin  | p                                           |                |
0 | 0 | 432 |          |          | In Pipeline 2 |                |
| |\          |-----+-----+-----+-----+-----+
| | +Filter      | cache[p.born] = $autoint_2                 |                |
0 | 0 |          |          |          | Fused in Pipeline 1 |                |
| | |          |-----+-----+-----+-----+-----+
| | +Expand(All) | (c)<-[anon_1:LIVES_IN]-(p)                 |                |
0 | 0 |          |          |          | Fused in Pipeline 1 |                |
| | |          |-----+-----+-----+-----+-----+
| | +Filter      | c:City                                       |                |
0 | 0 |          |          |          | Fused in Pipeline 1 |                |
| | |          |-----+-----+-----+-----+-----+
| | +Expand(All) | (cc)<-[anon_2:PART_OF]-(c)                 |                |
0 | 0 |          |          |          | Fused in Pipeline 1 |                |
| | |          |-----+-----+-----+-----+-----+
| | +NodeIndexSeek | BTREE INDEX cc:Country(formed) WHERE formed = $autoint_3 |                |
0 | 0 | 112 | 0/0 | 0.000 | Fused in Pipeline 1 |                |
| | |          |-----+-----+-----+-----+-----+
| | +Filter      | i.year = $autoint_1 AND cache[p.born] = $autoint_2 AND p:Pioneer |                |
0 | 1 |          |          |          | Fused in Pipeline 0 |                |
| | |          |-----+-----+-----+-----+-----+
| | +Expand(All) | (sc)<-[i:INVENTED_BY]-(p)                 |                |
1 | 3 |          |          |          | Fused in Pipeline 0 |                |
| | |          |-----+-----+-----+-----+-----+
| | +Filter      | sc:Science                                  |                |
1 | 1 |          |          |          | Fused in Pipeline 0 |                |
| | |          |-----+-----+-----+-----+-----+
| | +Expand(All) | (s)-[anon_0:RESEARCHED]->(sc)            |                |
1 | 2 |          |          |          | Fused in Pipeline 0 |                |
| | |          |-----+-----+-----+-----+-----+
| | +NodeIndexSeek | BTREE INDEX s:Scientist(born) WHERE born = $autoint_0 |                |
1 | 2 | 112 | 6/1 | 1.493 | Fused in Pipeline 0 |                |
+-----+-----+-----+-----+-----+-----+-----+

```

Total database accesses: 9, total allocated memory: 752

## 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

```
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

```
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

Operator	Details	Estimated Rows	Rows
DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)   Other
+ProduceResults	s, sc	1	1
0		2/0	0.224   In Pipeline 2
+NodeLeftOuterHashJoin	s	1	1
0	4864		11.193   In Pipeline 2
+Expand(All)	(sc)<-[anon_0:RESEARCHED]-(s)	100	100
300			Fused in Pipeline 1
+NodeByLabelScan	sc:Science	100	100
101	112	4/0	3.182   Fused in Pipeline 1
+NodeIndexSeek	BTREE INDEX s:Scientist(born) WHERE born = \$autoint_0	1	1
2	112	1/0	0.569   In Pipeline 0

Total database accesses: 403, total allocated memory: 4944

## PERIODIC COMMIT query hint Deprecated

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.

# Execution plans

This section describes the characteristics of query execution plans and provides details about each of the operators.



For information on replanning, see [Cypher replanning](#).

## 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](#). 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](#).

## 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 [Profile a query](#) 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](#) 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.

## Database hits

Each operator will send a request to the storage engine to do work such as retrieving or updating data. A database hit (DBHits) is an abstract unit of this storage engine work.

These are 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.

## Execution plan operators

*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.

Name	Description	Leaf?	Updating?	Considerations
<a href="#">AllNodesScan</a>	Reads all nodes from the node store.	Yes		
<a href="#">Anti</a>	Tests for the absence of a pattern.			
<a href="#">AntiSemiApply</a>	Performs a nested loop. Tests for the absence of a pattern predicate.			
<a href="#">Apply</a>	Performs a nested loop. Yields rows from both the left-hand and right-hand side operators.			
<a href="#">Argument</a>	Indicates the variable to be used as an argument to the right-hand side of an <a href="#">Apply</a> operator.	Yes		
<a href="#">AssertSameNode</a>	Used to ensure that no unique constraints are violated.			

Name	Description	Leaf?	Updating?	Considerations
<a href="#">AssertingMultiNodeIndexSeek</a>	Used to ensure that no unique constraints are violated.			
<a href="#">CacheProperties</a>	Reads node or relationship properties and caches them.			
<a href="#">CartesianProduct</a>	Produces a cartesian product of the inputs from the left-hand and right-hand operators.			
<a href="#">Create</a>	Creates nodes and relationships.		<input checked="" type="checkbox"/>	
<a href="#">CreateIndex</a>	Creates an index for either nodes or relationships.		<input checked="" type="checkbox"/>	
<a href="#">CreateNodeKeyConstraint</a>	Creates a node key constraint on a set of properties for all nodes with a certain label.		<input checked="" type="checkbox"/>	
<a href="#">CreateNodePropertyExistenceConstraint</a>	Creates an existence constraint on a property for all nodes with a certain label.		<input checked="" type="checkbox"/>	
<a href="#">CreateRelationshipPropertyExistenceConstraint</a>	Creates an existence constraint on a property for all relationships of a certain type.		<input checked="" type="checkbox"/>	
<a href="#">CreateUniqueConstraint</a>	Creates a unique constraint on a set of properties for all nodes with a certain label.		<input checked="" type="checkbox"/>	
<a href="#">Delete</a>	Deletes a node or relationship.		<input checked="" type="checkbox"/>	
<a href="#">DetachDelete</a>	Deletes a node and its relationships.		<input checked="" type="checkbox"/>	
<a href="#">DirectedRelationshipByIdSeek</a>	Reads one or more relationships by id from the relationship store.	<input checked="" type="checkbox"/>		

Name	Description	Leaf?	Updating?	Considerations
<a href="#">DirectedRelationshipIndexContainsScan</a>	Examines all values stored in an index, searching for entries containing a specific string; for example, in queries including <b>CONTAINS</b> .			
<a href="#">DirectedRelationshipIndexEndsWithScan</a>	Examines all values stored in an index, searching for entries ending in a specific string; for example, in queries containing <b>ENDS WITH</b> .			
<a href="#">DirectedRelationshipIndexScan</a>	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.			
<a href="#">DirectedRelationshipIndexSeek</a>	Finds relationships and their start and end nodes using an index seek.			
<a href="#">DirectedRelationshipIndexSeekByRange</a>	Finds relationships and their start and end nodes using an index seek where the value of the property matches a given prefix string.			
<a href="#">DirectedRelationshipTypeScan</a>	Fetches all relationships and their start and end nodes with a specific type from the relationship type index.			
<a href="#">Distinct</a>	Drops duplicate rows from the incoming stream of rows.			<b>Eager</b>
<a href="#">DoNothingIfExists(CONSTRAINT)</a>	Checks if a constraint already exists, if it does then it stops the execution, if not it continues.	<b>Yes</b>		
<a href="#">DoNothingIfExists(INDEX)</a>	Checks if an index already exists, if it does then it stops the execution, if not it continues.	<b>Yes</b>		
<a href="#">DropConstraint</a>	Drops a constraint using its name.	<b>Yes</b>	<b>Yes</b>	

Name	Description	Leaf?	Updating?	Considerations
DropIndex	Drops an index from a property for all nodes with a certain label.	Yes	Yes	Deprecated
DropIndex	Drops an index using its name.	Yes	Yes	
DropNodeKeyConstraint	Drops a node key constraint from a set of properties for all nodes with a certain label.	Yes	Yes	Deprecated
DropNodePropertyExistenceConstraint	Drops an existence constraint from a property for all nodes with a certain label.	Yes	Yes	Deprecated
DropRelationshipPropertyExistenceConstraint	Drops an existence constraint from a property for all relationships of a certain type.	Yes	Yes	Deprecated
DropUniqueConstraint	Drops a unique constraint from a set of properties for all nodes with a certain label.	Yes	Yes	Deprecated
Eager	For isolation purposes, <b>Eager</b> ensures that operations affecting subsequent operations are executed fully for the whole dataset before continuing execution.			Eager
EagerAggregation	Evaluates a grouping expression.			Eager
EmptyResult	Eagerly loads all incoming data and discards it.			
EmptyRow	Returns a single row with no columns.	Yes		
ExhaustiveLimit	The <b>ExhaustiveLimit</b> operator is similar to the <b>Limit</b> operator, but always exhausts the input. Used when combining <b>LIMIT</b> and updates.			

Name	Description	Leaf?	Updating?	Considerations
Expand(All)	Traverses incoming or outgoing relationships from a given node.			
Expand(Into)	Finds all relationships between two nodes.			
Filter	Filters each row coming from the child operator, only passing through rows that evaluate the predicates to <b>true</b> .			
Foreach	Performs a nested loop. Yields rows from the left-hand operator and discards rows from the right-hand operator.			
LetAntiSemiApply	Performs a nested loop. Tests for the absence of a pattern predicate in queries containing multiple pattern predicates.			
LetSelectOrAntiSemiApply	Performs a nested loop. Tests for the absence of a pattern predicate that is combined with other predicates.			
LetSelectOrSemiApply	Performs a nested loop. Tests for the presence of a pattern predicate that is combined with other predicates.			
LetSemiApply	Performs a nested loop. Tests for the presence of a pattern predicate in queries containing multiple pattern predicates.			
Limit	Returns the first <b>n</b> rows from the incoming input.			
LoadCSV	Loads data from a CSV source into the query.	Yes		

Name	Description	Leaf?	Updating?	Considerations
LockingMerge	Similar to the Merge operator but will lock the start and end node when creating a relationship if necessary.			
Merge	The Merge operator will either read or create nodes and/or relationships.			
MultiNodeIndexSeek	Finds nodes using multiple index seeks.	Yes		
NodeByIdSeek	Reads one or more nodes by ID from the node store.	Yes		
NodeByLabelScan	Fetches all nodes with a specific label from the node label index.	Yes		
NodeCountFromCountStore	Uses the count store to answer questions about node counts.	Yes		
NodeHashJoin	Executes a hash join on node ID.			Eager
NodeIndexContainsScan	Examines all values stored in an index, searching for entries containing a specific string.	Yes		
NodeIndexEndsWithScan	Examines all values stored in an index, searching for entries ending in a specific string.	Yes		
NodeIndexScan	Examines all values stored in an index, returning all nodes with a particular label with a specified property.	Yes		
NodeIndexSeek	Finds nodes using an index seek.	Yes		
NodeIndexSeekByRange	Finds nodes using an index seek where the value of the property matches the given prefix string.	Yes		



Name	Description	Leaf?	Updating?	Considerations
NodeLeftOuterHashJoin	Executes a left outer hash join.			Eager
NodeRightOuterHashJoin	Executes a right outer hash join.			Eager
NodeUniqueIndexSeek	Finds nodes using an index seek within a unique index.	Yes		
NodeUniqueIndexSeekByRange	Finds nodes using an index seek within a unique index where the value of the property matches the given prefix string.	Yes		
Optional	Yields a single row with all columns set to <code>null</code> if no data is returned by its source.			
OptionalExpand(All)	Traverses relationships from a given node, producing a single row with the relationship and end node set to <code>null</code> if the predicates are not fulfilled.			
OptionalExpand(Into)	Traverses all relationships between two nodes, producing a single row with the relationship and end node set to <code>null</code> if no matching relationships are found (the start node is the node with the smallest degree).			
OrderedAggregation	Like <code>EagerAggregation</code> but relies on the ordering of incoming rows. Is not eager.			
OrderedDistinct	Like <code>Distinct</code> but relies on the ordering of incoming rows.			
PartialSort	Sorts a row by multiple columns if there is already an ordering.			

Name	Description	Leaf?	Updating?	Considerations
PartialTop	Returns the first <i>n</i> rows sorted by multiple columns if there is already an ordering.			
ProcedureCall	Calls a procedure.			
ProduceResults	Prepares the result so that it is consumable by the user.			
ProjectEndpoints	Projects the start and end node of a relationship.			
Projection	Evaluates a set of expressions, producing a row with the results thereof.	Yes		
RelationshipCountFromCountStore	Uses the count store to answer questions about relationship counts.	Yes		
RemoveLabels	Deletes labels from a node.		Yes	
RollUpApply	Performs a nested loop. Executes a pattern expression or pattern comprehension.			
SelectOrAntiSemiApply	Performs a nested loop. Tests for the absence of a pattern predicate if an expression predicate evaluates to <i>false</i> .			
SelectOrSemiApply	Performs a nested loop. Tests for the presence of a pattern predicate if an expression predicate evaluates to <i>false</i> .			
SemiApply	Performs a nested loop. Tests for the presence of a pattern predicate.			
SetLabels	Sets labels on a node.		Yes	

Name	Description	Leaf?	Updating?	Considerations
SetNodePropertiesFromMap	Sets properties from a map on a node.		Yes	
SetProperty	Sets a property on a node or relationship.		Yes	
SetRelationshipPropertiesFromMap	Sets properties from a map on a relationship.		Yes	
ShortestPath	Finds one or all shortest paths between two previously matches node variables.			
ShowConstraints	Lists the available constraints.	Yes		
ShowFunctions	Lists the available functions.	Yes		
ShowIndexes	Lists the available indexes.	Yes		
ShowProcedures	Lists the available procedures.	Yes		
ShowTransactions	Lists the available transactions on the current server.	Yes		
Skip	Skips <i>n</i> rows from the incoming rows.			
Sort	Sorts rows by a provided key.			Eager
TerminateTransactions	Terminate transactions with the given IDs.	Yes		
Top	Returns the first 'n' rows sorted by a provided key.			Eager
TriadicBuild	The <b>TriadicBuild</b> operator is used in conjunction with <b>TriadicFilter</b> to solve triangular queries.			

Name	Description	Leaf?	Updating?	Considerations
<a href="#">TriadicFilter</a>	The <a href="#">TriadicFilter</a> operator is used in conjunction with <a href="#">TriadicBuild</a> to solve triangular queries.			
<a href="#">TriadicSelection</a>	Solves triangular queries, such as the very common 'find my friend-of-friends that are not already my friend'.			
<a href="#">UndirectedRelationshipByIdSeek</a>	Reads one or more relationships by ID from the relationship store.	Yes		
<a href="#">UndirectedRelationshipIndexContainsScan</a>	Examines all values stored in an index, searching for entries containing a specific string; for example, in queries including <a href="#">CONTAINS</a> .			
<a href="#">UndirectedRelationshipIndexEndsWithScan</a>	Examines all values stored in an index, searching for entries ending in a specific string; for example, in queries containing <a href="#">ENDS WITH</a> .			
<a href="#">UndirectedRelationshipIndexScan</a>	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.			
<a href="#">UndirectedRelationshipIndexSeek</a>	Finds relationships and their start and end nodes using an index seek.			
<a href="#">UndirectedRelationshipIndexSeekByRange</a>	Finds relationships and their start and end nodes using an index seek where the value of the property matches a given prefix string.			
<a href="#">UndirectedRelationshipTypeScan</a>	Fetches all relationships and their start and end nodes with a specific type from the relationship type index.			

Name	Description	Leaf?	Updating?	Considerations
<a href="#">Union</a>	Concatenates the results from the right-hand operator with the results from the left-hand operator.			
<a href="#">Unwind</a>	Returns one row per item in a list.			
<a href="#">ValueHashJoin</a>	Executes a hash join on arbitrary values.			<a href="#">Eager</a>
<a href="#">VarLengthExpand(All)</a>	Traverses variable-length relationships from a given node.			
<a href="#">VarLengthExpand(Into)</a>	Finds all variable-length relationships between two nodes.			
<a href="#">VarLengthExpand(Pruning)</a>	Traverses variable-length relationships from a given node and only returns unique end nodes.			

## Execution plan operators in detail

*All execution plan 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.

### 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.

## Example 319. AllNodesScan

### Query

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

### 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 | n      |                | 35 | 35 | 0 |          |
|                | Fused in Pipeline 0 |
| |              |
|                |
| +AllNodesScan  | n      |                | 35 | 35 | 36 |          |
3/0 | 0.598 | Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+

```

Total database accesses: 36, total allocated memory: 176

## 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.

## Example 320. DirectedRelationshipIndexScan

### Query

```
MATCH ()-[r: WORKS_IN]->()
WHERE r.title IS NOT NULL
RETURN r
```

### 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 | r |
| 15 | 15 | 0 | | | | Fused in |
Pipeline 0 |
| |
+-----+-----+-----+-----+-----+-----+-----+
| +DirectedRelationshipIndexScan | BTREE INDEX (anon_0)-[r:WORKS_IN(title)]->(anon_1) WHERE title IS NOT NULL |
| 15 | 15 | 31 | 112 | 3/1 | 0.874 |
Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+
```

Total database accesses: 31, total allocated memory: 176

## 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.

## Example 321. UndirectedRelationshipIndexScan

### Query

```
MATCH ()-[r: WORKS_IN]-()
WHERE r.title IS NOT NULL
RETURN r
```

### 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 | r | | | | | | Fused in |
| 30 | 30 | 0 | | | | | |
Pipeline 0 |
| |
+-----+-----+-----+-----+-----+-----+
| +UndirectedRelationshipIndexScan | BTREE INDEX (anon_0)-[r:WORKS_IN(title)]-(anon_1) WHERE title IS NOT NULL |
| 30 | 30 | 31 | 112 | 3/1 | 1.005 |
Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
```

Total database accesses: 31, total allocated memory: 176

## 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.



## Example 322. DirectedRelationshipIndexSeek

### Query

```
MATCH (candidate)-[r:WORKS_IN]->( )
WHERE r.title = 'chief architect'
RETURN candidate
```

### 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         | candidate |
| Pipeline 0 | 2 | 1 | 0 | | | Fused in |
+-----+-----+-----+-----+-----+-----+-----+
| +DirectedRelationshipIndexSeek | BTREE INDEX (candidate)-[r:WORKS_IN(title)]->(anon_0) WHERE title = $autostring_0 |
| 0.416 | Fused in Pipeline 0 | 2 | 1 | 3 | 112 | 3/1 |
+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+
```

Total database accesses: 3, total allocated memory: 176

## 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.

## Example 323. UndirectedRelationshipIndexSeek

### Query

```
MATCH (candidate)-[r:WORKS_IN]-()
WHERE r.title = 'chief architect'
RETURN candidate
```

### 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 | candidate |
| 4 | 2 | 0 | | | | Fused in |
Pipeline 0 |
| |
+-----+-----+-----+-----+-----+-----+
| +UndirectedRelationshipIndexSeek | BTREE INDEX (candidate)-[r:WORKS_IN(title)]-(anon_0) WHERE title = $autostring_0 |
| 4 | 2 | 3 | 112 | | 3/1 |
0.472 | Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
```

Total database accesses: 3, total allocated memory: 176

## 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.

## Example 324. DirectedRelationshipByIdSeek

### Query

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

### 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	r, n1	1	1			
0				Fused in Pipeline 0		
+DirectedRelationshipByIdSeek	(n1)-[r]->(anon_0) WHERE id(r) = \$autoint_0	1	1			
1	112	4/0	0.306	Fused in Pipeline 0		

Total database accesses: 1, total allocated memory: 176

## 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.

## Example 325. UndirectedRelationshipByIdSeek

### 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	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Other
+ProduceResults		r, n1			2
2	0				Fused in Pipeline 0
+UndirectedRelationshipByIdSeek		(n1)-[r]-(anon_0) WHERE id(r) = \$autoint_0			2
2	1	112	4/0	0.890	Fused in Pipeline 0

Total database accesses: 1, total allocated memory: 176

## 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.





## Example 328. DirectedRelationshipIndexEndsWithScan

### Query

```
MATCH ()-[r: WORKS_IN]->()
WHERE r.title ENDS WITH 'developer'
RETURN r
```

### 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 | r |
| 0 | 8 | 0 | | | | Fused in |
Pipeline 0 |
| |
+-----+-----+-----+-----+-----+-----+
+-----+
| +DirectedRelationshipIndexEndsWithScan | BTREE INDEX (anon_0)-[r:WORKS_IN(title)]->(anon_1) WHERE |
title ENDS WITH $autostring_0 | 0 | 8 | 17 | 112 |
3/1 | 1.158 | Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+-----+
+-----+
+-----+-----+-----+-----+-----+
+-----+
```

Total database accesses: 17, total allocated memory: 176

## 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.

## Example 329. UndirectedRelationshipIndexEndsWithScan

### Query

```
MATCH ()-[r: WORKS_IN]-()
WHERE r.title ENDS WITH 'developer'
RETURN r
```

### 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                         | r
|      0 | 16 |      0 |                |                |                | Fused in
Pipeline 0 |
| |
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
+-----+
| +UndirectedRelationshipIndexEndsWithScan | BTREE INDEX (anon_0)-[r:WORKS_IN(title)]-(anon_1) WHERE
title ENDS WITH $autostring_0 |      0 | 16 |      17 |      112 |
3/1 |      0.655 | Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
+-----+
+-----+-----+-----+-----+-----+
+-----+

Total database accesses: 17, total allocated memory: 176
```

## 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 `>=`.



## Example 330. DirectedRelationshipIndexSeekByRange

### Query

```
MATCH (candidate: Person)-[r:WORKS_IN]->(location)
WHERE r.duration > 100
RETURN candidate
```

### 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 | candidate | | | | | | Fused in |
Pipeline 0 | | | | | | | |
| | | | | | | |
+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+
| +Filter | candidate:Person | | | | | | Fused in |
Pipeline 0 | | | | | | | |
| | | | | | | |
+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+
| +DirectedRelationshipIndexSeekByRange | BTREE INDEX (candidate)-[r:WORKS_IN(duration)]->(location) |
WHERE duration > $autoint_0 | | 4 | 15 | 31 | | 112 |
4/1 | | 0.495 | Fused in Pipeline 0 | |
+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+
Total database accesses: 46, total allocated memory: 176
```

## 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 `>=`.

## Example 331. UndirectedRelationshipIndexSeekByRange

### Query

```
MATCH (candidate: Person)-[r:WORKS_IN]-(location)
WHERE r.duration > 100
RETURN candidate
```

### 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                               | candidate
|      4 | 15 |      0 |                |                |                | Fused in
Pipeline 0 |
| |
+-----+
+-----+-----+-----+-----+-----+
+-----+
| +Filter                                       | candidate:Person
|      4 | 15 |     30 |                |                |                | Fused in
Pipeline 0 |
| |
+-----+
+-----+-----+-----+-----+-----+
+-----+
| +UndirectedRelationshipIndexSeekByRange | BTREE INDEX (candidate)-[r:WORKS_IN(duration)]-(location)
WHERE duration > $autoint_0 |      8 | 30 |    31 |                | 112 |
4/1 |      0.728 | Fused in Pipeline 0 |
+-----+
+-----+
+-----+-----+-----+-----+-----+
+-----+
```

Total database accesses: 61, total allocated memory: 176

## 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.

### Example 332. DirectedRelationshipTypeScan

#### Query

```
MATCH ()-[r: FRIENDS_WITH]->()  
RETURN r
```

#### Query Plan

Compiler CYPHER 4.4

Planner COST

Runtime PIPELINED

Runtime version 4.4

Operator	Details	Estimated Rows	Rows	DB
+ProduceResults	r	2	2	
0				
Fused in Pipeline 0				
+DirectedRelationshipTypeScan	(anon_0)-[r:FRIENDS_WITH]->(anon_1)	2	2	
5	112   2/1   0.470			
Fused in Pipeline 0				

Total database accesses: 5, total allocated memory: 176

## 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

```
MATCH ()-[r: FRIENDS_WITH]-()  
RETURN r
```

## Query Plan

Compiler CYPHER 4.4

Planner COST

Runtime PIPELINED

Runtime version 4.4

Operator	Details	Estimated Rows	Rows	DB
+ProduceResults	r	4	4	
0	Fused in Pipeline 0			
+UndirectedRelationshipTypeScan	(anon_0)-[r:FRIENDS_WITH]-(anon_1)	4	4	
5	112   2/1   0.821   Fused in Pipeline 0			

Total database accesses: 5, total allocated memory: 176

## Node By Id Seek

The **NodeByIdSeek** operator reads one or more nodes by id from the node store.

## Example 333. NodeByIdSeek

### Query

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

### Query Plan

Compiler CYPHER 4.4

Planner COST

Runtime PIPELINED

Runtime version 4.4

Operator	Details	Estimated Rows	Rows	DB Hits	Memory (Bytes)
+ProduceResults	n	1	1	0	
	Fused in Pipeline 0				
+NodeByIdSeek	n WHERE id(n) = \$autoint_0	1	1	1	112
3/0	0.252 Fused in Pipeline 0				

Total database accesses: 1, total allocated memory: 176

## Node By Label Scan

The `NodeByLabelScan` operator fetches all nodes with a specific label from the node label index.

## Example 334. NodeByLabelScan

### Query

```
MATCH (person:Person)
RETURN 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	person	14	14	0				
	Fused in Pipeline 0							
+NodeByLabelScan	person:Person	14	14	15	112		0.446	
	2/1   Fused in Pipeline 0							

Total database accesses: 15, total allocated memory: 176

## 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**.

## Example 335. NodeIndexSeek

### Query

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

### 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 | location                                     | 1 | | | |
| 1 | 0 | | | | Fused in Pipeline 0 |
| | | | | | |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| +NodeIndexSeek | BTREE INDEX location:Location(name) WHERE name = $autostring_0 | 1 |
| 1 | 2 | 112 | 2/1 | 0.422 | Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

Total database accesses: 2, total allocated memory: 176

## 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.

## Example 336. NodeUniqueIndexSeek

### Query

```
MATCH (t:Team {name: 'Malmo'})
RETURN t
```

### Query Plan

Compiler CYPHER 4.4

Planner COST

Runtime PIPELINED

Runtime version 4.4

Operator	Details	Estimated Rows	Rows	DB
+ProduceResults	t	1	0	
+NodeUniqueIndexSeek	UNIQUE t:Team(name) WHERE name = \$autostring_0	1	0	

Total database accesses: 1, total allocated memory: 176

## 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)**.



### Example 337. MultiNodeIndexSeek

#### Query

```
CYPHER runtime=pipelined
MATCH
  (location:Location {name: 'Malmo'}),
  (person:Person {name: 'Bob'})
RETURN location, 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   | location, person                           |
1 | 1 | 0 | | | | Fused in Pipeline 0 |
| | | | | | | |
+-----+-----+-----+-----+-----+-----+-----+
+-----+
| +MultiNodeIndexSeek | BTREE INDEX location:Location(name) WHERE name = $autostring_0, |
1 | 0 | 0 | 112 | 2/2 | 1.216 | Fused in Pipeline 0 |
| | | | | | | |
| | | | | | | |
| | | | | | | |
+-----+-----+-----+-----+-----+-----+-----+
+-----+
+-----+
+-----+
+-----+
+-----+
+-----+
+-----+
+-----+
+-----+
+-----+

Total database accesses: 0, total allocated memory: 176
```

## 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.

## Example 338. AssertingMultiNodeIndexSeek

### Query

```
MERGE (t:Team {name: 'Engineering', id: 42})
```

### 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        |
| 1 | 0 | 0 | | | | Fused in
Pipeline 0 |
| |
+-----+
+-----+-----+-----+-----+-----+
+-----+
| +EmptyResult          |
| 1 | 0 | 0 | | | | Fused in
Pipeline 0 |
| |
+-----+
+-----+-----+-----+-----+-----+
+-----+
| +Merge                 | CREATE (t:Team {name: $autostring_0, id: $autoint_1})
| 1 | 1 | 0 | | | | Fused in
Pipeline 0 |
| |
+-----+
+-----+-----+-----+-----+-----+
+-----+
| +AssertingMultiNodeIndexSeek | UNIQUE t:Team(name) WHERE name = $autostring_0, UNIQUE t:Team(id)
WHERE id = $autoint_1 | 0 | 2 | 4 | 112 | 0/2 |
2.260 | Fused in Pipeline 0 |
+-----+
+-----+
+-----+-----+-----+-----+-----+
+-----+
```

Total database accesses: 4, total allocated memory: 176

## 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`.

### Example 339. NodeIndexSeekByRange

#### Query

```
MATCH (l:Location)
WHERE l.name STARTS WITH 'Lon'
RETURN l
```

#### 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 | 1 |
+-----+-----+-----+-----+-----+-----+
| 2 | 1 | 0 | | | Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
| +NodeIndexSeekByRange | BTREE INDEX l:Location(name) WHERE name STARTS WITH $autostring_0 |
+-----+-----+-----+-----+-----+-----+
| 2 | 1 | 2 | 112 | 3/0 | 1.095 | Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+

Total database accesses: 2, total allocated memory: 176
```

## 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`.

## Example 340. NodeUniqueIndexSeekByRange

### Query

```
MATCH (t:Team)
WHERE t.name STARTS WITH 'Ma'
RETURN t
```

### 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                         | t                                       |
2 | 0 | 0 |          |          |          |          | Fused in Pipeline 0 |
| |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
| +NodeUniqueIndexSeekByRange             | UNIQUE t:Team(name) WHERE name STARTS WITH $autostring_0 |
2 | 0 | 1 | 112 |          | 1/0 | 0.781 | Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
```

Total database accesses: 1, total allocated memory: 176

## 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.

## Example 341. NodeIndexContainsScan

### Query

```
MATCH (l:Location)
WHERE l.name CONTAINS 'al'
RETURN l
```

### 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  | | 1 | | | | | Fused in Pipeline 0 |
0 | 2 | 0 | | | | |
| | | | | | | |
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
| +NodeIndexContainsScan | BTREE INDEX l:Location(name) WHERE name CONTAINS $autostring_0 |
0 | 2 | 3 | 112 | 2/1 | 0.470 | Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
```

Total database accesses: 3, total allocated memory: 176

## 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.

## Example 342. NodeIndexEndsWithScan

### Query

```
MATCH (l:Location)
WHERE l.name ENDS WITH 'al'
RETURN l
```

### 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 | 1 | | | | | Fused in Pipeline 0 |
| | | | | | | | |
+-----+-----+-----+-----+-----+-----+-----+-----+
| +NodeIndexEndsWithScan | BTREE INDEX l:Location(name) WHERE name ENDS WITH $autostring_0 |
0 | 0 | 1 | 112 | 0/1 | 0.270 | Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
```

Total database accesses: 1, total allocated memory: 176

## Node Index Scan

The **NodeIndexScan** operator examines all values stored in an index, returning all nodes with a particular label and a specified property.

## Example 343. NodeIndexScan

### Query

```
MATCH (l:Location)
WHERE l.name IS NOT NULL
RETURN l
```

### 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  | 1                                           |                | 10 | 10 |
0 |                |                |                | Fused in Pipeline 0 |
| |                |                |                |                |
+-----+-----+-----+-----+-----+-----+-----+-----+
| +NodeIndexScan   | BTREE INDEX l:Location(name) WHERE name IS NOT NULL |                | 10 | 10 |
11 |                | 2/1 | 0.486 | Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
```

Total database accesses: 11, total allocated memory: 176

## 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.

## Example 344. Apply

### 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
+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
0	2152   0/0   0.173   Fused in Pipeline 1		
+NodeIndexSeek	BTREE INDEX p:Person(name) WHERE name = \$autostring_0	1	1
2	112   0/1   0.227   In Pipeline 0		

Total database accesses: 2, total allocated memory: 2216

## 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.



## Example 345. SemiApply

### Query

```
CYPHER runtime=slotted
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
+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)	2	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

## 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.

## Example 346. AntiSemiApply

### Query

```
CYPHER runtime=slotted
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

Operator	Details	Estimated Rows	Rows
DB Hits	Memory (Bytes)   Page Cache Hits/Misses		
+ProduceResults	`other.name`	4	13
0	0/0		
+Projection	other.name AS `other.name`	4	13
13	1/0		
+AntiSemiApply		4	13
0	0/0		
+Expand(Into)	(me)-[anon_2:FRIENDS_WITH]->(other)	0	0
55	888   28/0		
+Argument	me, other	14	14
0	0/0		
+CartesianProduct		14	14
0	0/0		
+Filter	other:Person	14	14
35	1/0		
+AllNodesScan	other	35	35
36	1/0		
+NodeIndexSeek	BTREE INDEX me:Person(name) WHERE name = \$autostring_0	1	1
2	0/1		

Total database accesses: 141, total allocated memory: 968

## 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.

## Example 347. Anti

### 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   0.086   In Pipeline 4		
+Projection	other.name AS 'other.name'	4	13
26	2/0   0.069   In Pipeline 4		
+Apply		4	13
0	0/0		
+Anti		14	13
0	1256   0/0   0.070   In Pipeline 4		
+Limit	1	14	1
0	752   Fused in Pipeline 3		
+Expand(Into)	(me)-[anon_2:FRIENDS_WITH]->(other)	0	1
55	2848   Fused in Pipeline 3		
+Argument	me, other	14	14
0	3184   1/0   1.158   Fused in Pipeline 3		
+CartesianProduct		14	14
0	3664   0.107   In Pipeline 2		
+Filter	other:Person	14	14
0	Fused in Pipeline 1		
+AllNodesScan	other	35	35
36	128   1/0   0.158   Fused in Pipeline 1		
+NodeIndexSeek	BTREE INDEX me:Person(name) WHERE name = \$autostring_0	1	1
2	112   0/1   0.168   In Pipeline 0		

Total database accesses: 119, total allocated memory: 6736

---

## 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.

## Example 348. LetSemiApply

### Query

```
CYPHER runtime=slotted
MATCH (other:Person)
WHERE (other)-[:FRIENDS_WITH]->(:Person) OR (other)-[:WORKS_IN]->(:Location)
RETURN other.name
```

## Query Plan

Compiler CYPHER 4.4

Planner COST

Runtime SLOTTED

Runtime version 4.4

Operator	Details	Estimated Rows	Rows	DB Hits
+ProduceResults	`other.name`	13	14	0
+Projection	other.name AS `other.name`	13	14	14
+SelectOrSemiApply	anon_9	14	14	0
+Filter	anon_7:Location	14	0	12
+Expand(All)	(other)-[anon_6:WORKS_IN]->(anon_7)	14	12	26
+Argument	other	14	12	0
+LetSemiApply		14	14	0
+Filter	anon_5:Person	2	0	2
+Expand(All)	(other)-[anon_4:FRIENDS_WITH]->(anon_5)	2	2	33
+Argument	other	14	14	0
+Filter	other:Person	14	14	35
+AllNodesScan	other	35	35	36

Total database accesses: 158, total allocated memory: 64

## 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.



## Example 349. LetAntiSemiApply

### Query

```
CYPHER runtime=slotted
MATCH (other:Person)
WHERE NOT ((other)-[:FRIENDS_WITH]->(:Person)) OR (other)-[:WORKS_IN]->(:Location)
RETURN other.name
```

## Query Plan

Compiler CYPHER 4.4

Planner COST

Runtime SLOTTED

Runtime version 4.4

Operator	Details	Estimated Rows	Rows	DB Hits
+ProduceResults	`other.name`	11	14	0
+Projection	other.name AS `other.name`	11	14	14
+SelectOrSemiApply	anon_9	14	14	0
+Filter	anon_7:Location	14	0	2
+Expand(All)	(other)-[anon_6:WORKS_IN]->(anon_7)	14	2	7
+Argument	other	14	2	0
+LetAntiSemiApply		14	14	0
+Filter	anon_5:Person	2	0	2
+Expand(All)	(other)-[anon_4:FRIENDS_WITH]->(anon_5)	2	2	33
+Argument	other	14	14	0
+Filter	other:Person	14	14	35
+AllNodesScan	other	35	35	36

Total database accesses: 129, total allocated memory: 64

## Select Or Semi Apply

The **SelectOrSemiApply** operator tests for the presence of a pattern predicate and evaluates a predicate, and is a variation of the **Apply** operator. This operator allows for the mixing of normal predicates and

pattern predicates that check for the presence of a pattern. First, the normal expression predicate is evaluated, and, only if it returns **false**, is the costly pattern predicate evaluated.

### Example 350. SelectOrSemiApply

#### 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` Fused in Pipeline 2	11	2	0
+Projection	other.name AS `other.name` Fused in Pipeline 2	11	2	4
+SelectOrSemiApply	other.age > \$autoint_0 4200   0/0   0.153   Fused in Pipeline 2	14	2	0
+Limit	1 752   Fused in Pipeline 1	14	2	0
+Filter	anon_3:Person Fused in Pipeline 1	2	2	2
+Expand(All)	(other)-[anon_2:FRIENDS_WITH]->(anon_3) Fused in Pipeline 1	2	2	32
+Argument	other 2160   2/0   0.376   Fused in Pipeline 1	14	14	0
+Filter	other:Person Fused in Pipeline 0	14	14	0
+AllNodesScan	other 112   1/0   0.166   Fused in Pipeline 0	35	35	36

Total database accesses: 74, total allocated memory: 5040

## Select Or Anti Semi Apply

The `SelectOrAntiSemiApply` operator is used to evaluate **OR** between a predicate and a negative pattern predicate (i.e. a pattern predicate preceded with **NOT**), and is a variation of the `Apply` operator. If the predicate returns `true`, the pattern predicate is not tested. If the predicate returns `false` or `null`, `SelectOrAntiSemiApply` will instead test the pattern predicate.

## Example 351. SelectOrAntiSemiApply

### Query

```
MATCH (other:Person)
WHERE other.age > 25 OR NOT (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`	4	12	0
	Fused in Pipeline 3			
+Projection	other.name AS `other.name`	4	12	24
	Fused in Pipeline 3			
+SelectOrAntiSemiApply	other.age > \$autoint_0	14	12	0
	4200   0/0   0.215   Fused in Pipeline 3			
+Anti		14	12	0
	1256   0/0   0.127   In Pipeline 2			
+Limit	1	14	2	0
	752   Fused in Pipeline 1			
+Filter	anon_3:Person	2	2	2
	Fused in Pipeline 1			
+Expand(All)	(other)-[anon_2:FRIENDS_WITH]->(anon_3)	2	2	32
	Fused in Pipeline 1			
+Argument	other	14	14	0
	2160   2/0   1.109   Fused in Pipeline 1			
+Filter	other:Person	14	14	0
	Fused in Pipeline 0			
+AllNodesScan	other	35	35	36
	112   1/0   0.160   Fused in Pipeline 0			

Total database accesses: 94, total allocated memory: 5736

## 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.

## Example 352. LetSelectOrSemiApply

### 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
+ProduceResults	`other.name`	13	14	0
+Projection	other.name AS `other.name`	13	14	14
+SelectOrSemiApply	anon_9	14	14	0
+Filter	anon_7:Location	14	0	12
+Expand(All)	(other)-[anon_6:WORKS_IN]->(anon_7)	14	12	26
+Argument	other	14	12	0
+LetSelectOrSemiApply	other.age = \$autoint_0	14	14	14
+Filter	anon_5:Person	2	0	2
+Expand(All)	(other)-[anon_4:FRIENDS_WITH]->(anon_5)	2	2	33
+Argument	other	14	14	0
+Filter	other:Person	14	14	35
+AllNodesScan	other	35	35	36

Total database accesses: 172, total allocated memory: 64

## 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.

### Example 353. LetSelectOrAntiSemiApply

#### Query

```
CYPHER runtime=slotted
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

Operator	Details	Estimated Rows	Rows	DB
0	+ProduceResults   `other.name`	12	14	
14	+Projection   other.name AS `other.name`	12	14	
0	+SelectOrSemiApply   anon_9	14	14	
2	+Filter   anon_7:Location	14	0	
7	+Expand(All)   (other)-[anon_6:WORKS_IN]->(anon_7)	14	2	
0	+Argument   other	14	2	
14	+LetSelectOrAntiSemiApply   other.age = \$autoint_0	14	14	
2	+Filter   anon_5:Person	2	0	
33	+Expand(All)   (other)-[anon_4:FRIENDS_WITH]->(anon_5)	2	2	
0	+Argument   other	14	14	
35	+Filter   other:Person	14	14	
36	+AllNodesScan   other	35	35	

Total database accesses: 143, total allocated memory: 64

## 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.

### Example 354. Merge

#### 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 Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Other
|
+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+
| +ProduceResults |
1 | 0 | 0 | | | Fused in Pipeline 0 |
| |
+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+
| +EmptyResult   |
1 | 0 | 0 | | | Fused in Pipeline 0 |
| |
+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+
| +Merge         | CREATE (p:Person {name: $autostring_0}), ON MATCH SET p.existed = true, |
1 | 1 | 2 | | | Fused in Pipeline 0 |
| | | ON CREATE SET p.existed = false
| | |
| | |
+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+
| +NodeIndexSeek | BTREE INDEX p:Person(name) WHERE name = $autostring_0
1 | 1 | 2 | 112 | 2/1 | 0.512 | Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+

```

Total database accesses: 4, total allocated memory: 176

## 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.

## Example 355. LockingMerge

### 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

Operator	Details	Estimated Rows	Rows
+ProduceResults		1	0
0	Fused in Pipeline 1		
+EmptyResult		1	0
0	Fused in Pipeline 1		
+Apply		1	1
0			
+LockingMerge	CREATE (s)-[anon_0:FRIENDS_WITH]->(s), LOCK(s)	1	1
1	Fused in Pipeline 1		
+Expand(Into)	(s)-[anon_0:FRIENDS_WITH]->(s)	0	0
8	888 Fused in Pipeline 1		
+Argument	s	1	3
0	2152 3/0 0.616 Fused in Pipeline 1		
+NodeIndexSeek	BTREE INDEX s:Person(name) WHERE name = \$autostring_0	1	1
2	112 0/1 0.235 In Pipeline 0		

Total database accesses: 11, total allocated memory: 2232

## 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.

## Example 356. RollUpApply

### Query

```
CYPHER runtime=slotted
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

Operator	Details	Estimated Rows	Rows	DB Hits	Page Cache
+ProduceResults	`p.name`, cities	14	14	0	
+Projection	p.name AS `p.name`	14	14	14	
+RollUpApply	cities, anon_0	14	14	0	
+Projection	location.name AS anon_0	6	15	15	
+Expand(All)	(p)-[anon_2:WORKS_IN]->(location)	6	15	33	
+Argument	p	14	14	0	
+Filter	p:Person	14	14	35	
+AllNodesScan	p	35	35	36	

Total database accesses: 133, total allocated memory: 64

## Argument

The **Argument** operator indicates the variable to be used as an argument to the right-hand side of an **Apply** operator.

## Example 357. Argument

### 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

Operator	Details	Estimated Rows	Rows
+ProduceResults		1	0
0	Fused in Pipeline 1		
+EmptyResult		1	0
0	Fused in Pipeline 1		
+Apply		1	1
0			
+LockingMerge	CREATE (s)-[anon_0:FRIENDS_WITH]->(s), LOCK(s)	1	1
1	Fused in Pipeline 1		
+Expand(Into)	(s)-[anon_0:FRIENDS_WITH]->(s)	0	0
8	888 Fused in Pipeline 1		
+Argument	s	1	3
0	2152 3/0 0.894 Fused in Pipeline 1		
+NodeIndexSeek	BTREE INDEX s:Person(name) WHERE name = \$autostring_0	1	1
2	112 0/1 1.055 In Pipeline 0		

Total database accesses: 11, total allocated memory: 2232

## Expand All

Given a start node, and depending on the pattern relationship, the `Expand(All)` operator will traverse incoming or outgoing relationships.

## Example 358. Expand(All)

### Query

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

### Query Plan

Compiler CYPHER 4.4

Planner COST

Runtime PIPELINED

Runtime version 4.4

Operator	Details	Estimated Rows	Rows
+ProduceResults	fof	0	1
+Expand(All)	(p)-[anon_0:FRIENDS_WITH]->(fof)	0	1
+NodeIndexSeek	BTREE INDEX p:Person(name) WHERE name = \$autostring_0	1	1

Total database accesses: 5, total allocated memory: 176

## 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.



## Example 359. Expand(Into)

### 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
+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	Fused in Pipeline 0		
+Expand(All)	(p)<-[anon_1]- (fof)	0	0
3	Fused in Pipeline 0		
+NodeIndexSeek	BTREE INDEX p:Person(name) WHERE name = \$autostring_0	1	1
2	112   2/1   0.314   Fused in Pipeline 0		

Total database accesses: 5, total allocated memory: 192

## 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`.

## Example 360. OptionalExpand(All)

### Query

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

### 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, l                                       |
14 | 15 | 1 |          |          |          | Fused in Pipeline 0 |
| |
+-----+-----+-----+-----+-----+
+-----+
| +OptionalExpand(All) | (p)-[works_in:WORKS_IN]->(l) WHERE works_in.duration > $autoint_0 |
14 | 15 | 33 |          |          |          | Fused in Pipeline 0 |
| |
+-----+-----+-----+-----+-----+
+-----+
| +Filter          | p:Person                                   |
14 | 14 | 0 |          |          |          | Fused in Pipeline 0 |
| |
+-----+-----+-----+-----+-----+
+-----+
| +AllNodesScan    | p                                          |
35 | 35 | 36 | 112 | 5/0 | 0.764 | Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+
+-----+
+-----+
```

Total database accesses: 70, total allocated memory: 176

## 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.

## Example 361. OptionalExpand(Into)

### Query

```
MATCH (p:Person)-[works_in:WORKS_IN]->(1)
OPTIONAL MATCH (1)-->(p)
RETURN p
```

### 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   Fused in Pipeline 0	15	15	0	
+OptionalExpand(Into)	(1)-[anon_0]->(p)   Fused in Pipeline 0	15	15	105	3352
+Expand(All)	(p)-[works_in:WORKS_IN]->(1)   Fused in Pipeline 0	15	15	19	
+Filter	p:Person   Fused in Pipeline 0	14	14	0	
+AllNodesScan	p   Fused in Pipeline 0	35	35	36	112
	6/0   1.131   Fused in Pipeline 0				

Total database accesses: 160, total allocated memory: 3432

## VarLength Expand All

Given a start node, the `VarLengthExpand(All)` operator will traverse variable-length relationships.

## Example 362. VarLengthExpand(All)

### Query

```
MATCH (p:Person)-[:FRIENDS_WITH *1..2]-(q:Person)
RETURN p, q
```

### Query Plan

Compiler CYPHER 4.4

Planner COST

Runtime PIPELINED

Runtime version 4.4

Operator	Details	Estimated Rows	Rows	DB Hits
+ProduceResults	p, q Fused in Pipeline 0	4	6	0
+Filter	q:Person Fused in Pipeline 0	4	6	6
+VarLengthExpand(All)	(p)-[anon_0:FRIENDS_WITH*.2]-(q) Fused in Pipeline 0	4	6	47
+Filter	p:Person Fused in Pipeline 0	14	14	0
+AllNodesScan	p 7/0   1.033   Fused in Pipeline 0	35	35	36

Total database accesses: 89, total allocated memory: 208

## 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.

## Example 363. VarLengthExpand(Into)

### Query

```
MATCH (p:Person)-[:FRIENDS_WITH *1..2]-(p:Person)
RETURN p
```

### 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   Fused in Pipeline 0	0	0	0
+VarLengthExpand(Into)	(p)-[anon_0:FRIENDS_WITH*.2]-(p)   Fused in Pipeline 0	0	0	47
+Filter	p:Person   Fused in Pipeline 0	14	14	0
+AllNodesScan	p   4/0   0.622   Fused in Pipeline 0	35	35	36

Total database accesses: 83, total allocated memory: 192

## 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.

## Example 364. VarLengthExpand(Pruning)

### Query

```
MATCH (p:Person)-[:FRIENDS_WITH *3..4]-(q:Person)
RETURN DISTINCT p, q
```

### Query Plan

Compiler CYPHER 4.4

Planner COST

Runtime PIPELINED

Runtime version 4.4

Operator (Bytes)	Page Cache Hits/Misses	Details	Time (ms)	Other	Estimated Rows	Rows	DB Hits	Memory
+ProduceResults	0/0	p, q	0.049	In Pipeline 1	0	0	0	
+Distinct	0/0	p, q	4.258	In Pipeline 1	0	0	0	
+Filter	0/0	q:Person	0.165	In Pipeline 1	0	0	0	
+VarLengthExpand(Pruning)		(p)-[:FRIENDS_WITH*3..4]-(q)		In Pipeline 1	0	0	32	
+Filter		p:Person		Fused in Pipeline 0	14	14	0	
+AllNodesScan	1/0	p	0.224	Fused in Pipeline 0	35	35	36	

Total database accesses: 68, total allocated memory: 1208

## 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.

## Example 365. AssertSameNode

### Query

```
CYPHER runtime=slotted
MERGE (t:Team {name: 'Engineering', id: 42})
```

### Query Plan

Compiler CYPHER 4.4

Planner COST

Runtime SLOTTED

Runtime version 4.4

Operator	Details	Estimated
+ProduceResults		
1   0   0	0/0	
+EmptyResult		
1   0   0	0/0	
+Merge	CREATE (t:Team {name: \$autostring_0, id: \$autoint_1})	
1   1   0	0/0	
+AssertSameNode	t	
0   1   0	0/0	
+NodeUniqueIndexSeek(Locking)	UNIQUE t:Team(id) WHERE id = \$autoint_1	
1   1   1	0/1	
+NodeUniqueIndexSeek(Locking)	UNIQUE t:Team(name) WHERE name = \$autostring_0	
1   1   1	0/1	

Total database accesses: 2, total allocated memory: 64

## Empty Result

The `EmptyResult` operator eagerly loads all incoming data and discards it.

## Example 366. EmptyResult

### Query

```
CREATE (:Person)
```

### Query Plan

Compiler CYPHER 4.4

Planner COST

Runtime PIPELINED

Runtime version 4.4

Operator (ms)   Other	Details	Estimated Rows	Rows	DB Hits	Page Cache Hits/Misses	Time
+ProduceResults		1	0	0		
Fused in Pipeline 0						
+EmptyResult		1	0	0		
Fused in Pipeline 0						
+Create   (anon_0:Person)		1	1	1		0/0
0.000   Fused in Pipeline 0						

Total database accesses: 1, total allocated memory: 176

## 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.



## Example 367. ProduceResults

### Query

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

### 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 | n      |          35 | 35 |    0 |                |
|               | Fused in Pipeline 0 |
| |             |
|               |
| +AllNodesScan | n      |          35 | 35 |   36 |          112 |
3/0 |    0.853 | Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+

```

Total database accesses: 36, total allocated memory: 176

## 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.

## Example 368. LoadCSV

### 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 |    0.414 | In Pipeline 1 |      |         |                 |
| |              |       |               |      |         |                 |
+-----+-----+-----+-----+-----+
| +LoadCSV       | line   |           10 |    4 |    0 |                 | 104 |
|               | In Pipeline 1 |      |         |         |                 |
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+

```

Total database accesses: 0, total allocated memory: 176

## 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](#)

## 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.

## Example 369. NodeHashJoin

### 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
+ProduceResults	`loc.name`	10	0
+Projection	loc.name AS `loc.name`	10	0
+Filter	not anon_0 = anon_1	10	0
+NodeHashJoin	loc	10	0
+Expand(All)	(matt)-[anon_1:WORKS_IN]->(loc)	19	0
+NodeIndexSeek	BTREE INDEX matt:Person(name) WHERE name = \$autostring_1	1	0
+Expand(All)	(bob)-[anon_0:WORKS_IN]->(loc)	19	1
+NodeIndexSeek	BTREE INDEX bob:Person(name) WHERE name = \$autostring_0	1	1

Total database accesses: 6, total allocated memory: 3872

## 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).

## Example 370. ValueHashJoin

### 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

Operator	Details	Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache
+ProduceResults	p, q	10	0	0		
0/0	0.000   In Pipeline 2					
+ValueHashJoin	p.age = q.age	10	0	0	344	
	In Pipeline 2					
+Filter	q:Person	14	0	0		
	Fused in Pipeline 1					
+AllNodesScan	q	35	0	0	112	
0/0	0.000   Fused in Pipeline 1					
+Filter	p:Person	14	14	0		
	Fused in Pipeline 0					
+AllNodesScan	p	35	35	36	112	
1/0	0.340   Fused in Pipeline 0					

Total database accesses: 36, total allocated memory: 648

## Node Left/Right Outer Hash Join

The `NodeLeftOuterHashJoin` and `NodeRightOuterHashJoin` operators are variations of the `hash join`. The query below can be planned with either a left or a right outer join. The decision depends on the cardinalities of the left-hand and right-hand sides; i.e. how many rows would be returned, respectively, for `(a:Person)` and `(a)->(b:Person)`. If `(a:Person)` returns fewer results than `(a)->(b:Person)`, a left outer join — indicated by `NodeLeftOuterHashJoin` — is planned. On the other hand, if `(a:Person)` returns more results than `(a)->(b:Person)`, a right outer join — indicated by `NodeRightOuterHashJoin` — is planned instead.

## Example 371. NodeRightOuterHashJoin

### Query

```
MATCH (a:Person)
OPTIONAL MATCH (a)-->(b:Person)
USING JOIN ON a
RETURN a.name, b.name
```

### Query Plan

Compiler CYPHER 4.4

Planner COST

Runtime PIPELINED

Runtime version 4.4

Operator	Details	Estimated Rows
+ProduceResults	`a.name`, `b.name`	14
14   0	0/0   0.122   In Pipeline 2	
+Projection	cache[a.name] AS `a.name`, cache[b.name] AS `b.name`	14
14   24	0/0   0.287   In Pipeline 2	
+NodeRightOuterHashJoin	a	14
14   0	6176   0.377   In Pipeline 2	
+Filter	a:Person	14
14   0	Fused in Pipeline 1	
+AllNodesScan	a	35
35   36	112   0/0   0.170   Fused in Pipeline 1	
+CacheProperties	cache[a.name], cache[b.name]	2
2   6	Fused in Pipeline 0	
+Expand(All)	(b)<-[anon_0]-(a)	2
2   19	Fused in Pipeline 0	
+Filter	b:Person	14
14   0	Fused in Pipeline 0	
+AllNodesScan	b	35
35   36	112   4/0   0.347   Fused in Pipeline 0	

Total database accesses: 121, total allocated memory: 6264

## 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.

### Example 372. TriadicSelection

#### Query

```
CYPHER runtime=slotted
MATCH (me:Person)-[:FRIENDS_WITH]-()-[:FRIENDS_WITH]-(other)
WHERE NOT (me)-[:FRIENDS_WITH]-(other)
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`	0	2	0	0/0
+Projection	other.name AS `other.name`	0	2	2	1/0
+TriadicSelection	WHERE NOT (me)--(other)	0	2	0	0/0
+Filter	not anon_2 = anon_4	0	2	0	0/0
+Expand(All)	(anon_3)-[anon_4:FRIENDS_WITH]-(other)	0	6	14	8/0
+Argument	anon_3, anon_2	4	4	0	0/0
+Expand(All)	(me)-[anon_2:FRIENDS_WITH]-(anon_3)	4	4	33	28/0
+Filter	me:Person	14	14	35	1/0
+AllNodesScan	me	35	35	36	1/0

Total database accesses: 120, total allocated memory: 64

## 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.

## Example 373. TriadicBuild

### Query

```
CYPHER runtime=pipelined
MATCH (me:Person)-[:FRIENDS_WITH]-()-[: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

Operator (Bytes)	Details Page Cache Hits/Misses	Time (ms)   Other	Estimated Rows	Rows	DB Hits	Memory
+ProduceResults	`other.name` 0/0	0.121   In Pipeline 3	0	2	0	
+Projection	other.name AS `other.name` 2/0	0.092   In Pipeline 3	0	2	4	
+TriadicFilter	WHERE NOT (me)--(other) 6984   0/0	0.122   In Pipeline 3	0	2	0	
+Apply	0/0		0	2	0	
+Filter	not anon_2 = anon_4   Fused in Pipeline 2		0	2	0	
+Expand(All)	(anon_3)-[anon_4:FRIENDS_WITH]-(other)   Fused in Pipeline 2		0	6	14	
+Argument	anon_3, anon_2 4200   0/0	0.397   Fused in Pipeline 2	4	4	0	
+TriadicBuild	(me)--(anon_3) 3392   0/0	0.481   In Pipeline 1	4	4	0	
+Expand(All)	(me)-[anon_2:FRIENDS_WITH]-(anon_3)   Fused in Pipeline 0		4	4	19	
+Filter	me:Person   Fused in Pipeline 0		14	14	0	
+AllNodesScan	me 112   2/0	0.584   Fused in Pipeline 0	35	35	36	

Total database accesses: 73, total allocated memory: 7248



## 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.

## Example 374. TriadicFilter

### Query

```
CYPHER runtime=pipelined
MATCH (me:Person)-[:FRIENDS_WITH]-()-[: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

Operator (Bytes)	Details Page Cache Hits/Misses	Time (ms)   Other	Estimated Rows	Rows	DB Hits	Memory
+ProduceResults	`other.name` 0/0	0.061   In Pipeline 3	0	2	0	
+Projection	other.name AS `other.name` 2/0	0.148   In Pipeline 3	0	2	4	
+TriadicFilter	WHERE NOT (me)--(other) 6984   0/0	0.411   In Pipeline 3	0	2	0	
+Apply	0/0		0	2	0	
+Filter	not anon_2 = anon_4   Fused in Pipeline 2		0	2	0	
+Expand(All)	(anon_3)-[anon_4:FRIENDS_WITH]-(other)   Fused in Pipeline 2		0	6	14	
+Argument	anon_3, anon_2 4200   0/0	0.362   Fused in Pipeline 2	4	4	0	
+TriadicBuild	(me)--(anon_3) 3392   0/0	3.763   In Pipeline 1	4	4	0	
+Expand(All)	(me)-[anon_2:FRIENDS_WITH]-(anon_3)   Fused in Pipeline 0		4	4	19	
+Filter	me:Person   Fused in Pipeline 0		14	14	0	
+AllNodesScan	me 112   2/0	0.279   Fused in Pipeline 0	35	35	36	

Total database accesses: 73, total allocated memory: 7248

# 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.

## Example 375. CartesianProduct

### Query

```
MATCH
  (p:Person),
  (t:Team)
RETURN p, t
```

### 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, t    | 140            | 140 | 0       |                 |
2/0 | 1.868 | In Pipeline 2 |      |         |                 |
| |               +-----+-----+-----+-----+
| +CartesianProduct |         | 140            | 140 | 0       | 3656 |
| 0.528 | In Pipeline 2 |         |      |         |                 |
| |\            +-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
| | +Filter        | t:Team  | 10            | 10 | 0       |
| |               | Fused  |                 |     |         |
| |               | in Pipeline 1 |
| |               +-----+-----+-----+-----+
| | +AllNodesScan | t       | 35            | 35 | 36      | 128 |
0/0 | 0.123 | Fused in Pipeline 1 |
| |               +-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
| +Filter          | p:Person | 14            | 14 | 0       | |
| |               | Fused  |                 |     |         |
| |               | in Pipeline 0 |
| |               +-----+-----+-----+-----+
| |               +-----+-----+-----+-----+
| +AllNodesScan   | p       | 35            | 35 | 36      | 112 |
1/0 | 0.164 | Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+

Total database accesses: 72, total allocated memory: 3736
```

# 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.

### Example 376. Foreach

#### Query

```
CYPHER runtime=slotted
FOREACH (value IN [1,2,3] | CREATE (:Person {age: value}))
```

#### Query Plan

Compiler CYPHER 4.4

Planner COST

Runtime SLOTTED

Runtime version 4.4

Operator	Details	Estimated Rows	Rows
+ProduceResults		1	0
+EmptyResult		1	0
+Foreach	value IN [1, 2, 3], CREATE (anon_0:Person {age: value})	1	1

Total database accesses: 9, total allocated memory: 64

## 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.

## Example 377. Eager

### Query

```
MATCH (a)-[r]-(b)
DELETE r, a, b
MERGE ()
```

## 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
+ProduceResults	Fused in Pipeline 3	36	0	0		
+EmptyResult	Fused in Pipeline 3	36	0	0		
+Apply		36	504	0		
+Merge	CREATE (anon_0) Fused in Pipeline 3	36	504	0		
+AllNodesScan	anon_0 Fused in Pipeline 3	1260	504	540	32872	0/0   0.794
+Eager	In Pipeline 2	36	36	0	24696	0/0   0.081
+Delete	b Fused in Pipeline 1	36	36	9		
+Delete	a Fused in Pipeline 1	36	36	12		
+Delete	r Fused in Pipeline 1	36	36	18		
+Eager	delete overlap: b, r Fused in Pipeline 1	36	36	0	24696	2/0   1.629
+Expand(All)	(a)-[r]-(b) Fused in Pipeline 0	36	36	36		
+AllNodesScan	a Fused in Pipeline 0	35	35	36	112	2/0   0.245

Total database accesses: 651, total allocated memory: 33000

## 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.

### Example 378. EagerAggregation

#### 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	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Other	Estimated Rows	Rows
+ProduceResults	location, people	0		0/0	0.138	In Pipeline 1	4	6
+EagerAggregation	cache[l.name] AS location, collect(p.name) AS people	30	3168			Fused in Pipeline 0	4	6
+Filter	p:Person	15				Fused in Pipeline 0	15	15
+Expand(All)	(l)<-[anon_0:WORKS_IN]-(p)	16				Fused in Pipeline 0	15	15
+CacheProperties	cache[l.name]	10				Fused in Pipeline 0	10	10
+Filter	l:Location	0				Fused in Pipeline 0	10	10
+AllNodesScan	l	36	112	4/0	0.484	Fused in Pipeline 0	35	35

Total database accesses: 107, total allocated memory: 3248

## 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.

## Example 379. OrderedAggregation

### Query

```
MATCH (p:Person)
WHERE p.name STARTS WITH 'P'
RETURN p.name, count(*) AS count
```

### 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) | Ordered by |
| Other          |       |         |                 |                         |           |           |
+-----+-----+-----+-----+-----+-----+
| +ProduceResults  | `p.name`, count | | | | | | |
|           0 | 2 | 0 |           |           | 0/0 | 0.060 | p.name ASC |
| In Pipeline 1 | |
| |
+-----+-----+-----+-----+-----+-----+
| +OrderedAggregation | cache[p.name] AS `p.name`, count(*) AS count | | | | | | |
|           0 | 2 | 0 | 1832 |           | 0/0 | 0.154 | p.name ASC |
| In Pipeline 1 | |
| |
+-----+-----+-----+-----+-----+-----+
| +NodeIndexSeekByRange | BTREE INDEX p:Person(name) WHERE name STARTS WITH $autostring_0, | | | | | |
| cache[p.name] | 0 | 2 | 3 | 112 | 0/1 | 0.711 |
| p.name ASC | In Pipeline 0 |
+-----+-----+-----+-----+-----+-----+
Total database accesses: 3, total allocated memory: 1896
```

## 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.



## Example 380. NodeCountFromCountStore

### Query

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

### Query Plan

Compiler CYPHER 4.4

Planner COST

Runtime PIPELINED

Runtime version 4.4

Operator (Bytes)	Details Page Cache Hits/Misses	Time (ms)	Other	Estimated Rows	Rows	DB Hits	Memory
+ProduceResults	people			1	1	0	
			Fused in Pipeline 0				
+NodeCountFromCountStore	count( (:Person) ) AS people			1	1	1	
112	0/0	0.187	Fused in Pipeline 0				

Total database accesses: 1, total allocated memory: 176

## 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.

## Example 381. RelationshipCountFromCountStore

### Query

```
MATCH (p:Person)-[r:WORKS_IN]->()
RETURN count(r) AS jobs
```

### 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	jobs	1
1	0       Fused in Pipeline 0	
+RelationshipCountFromCountStore	count( (:Person)-[:WORKS_IN]->() ) AS jobs	1
1	1   112   0/0   0.159   Fused in Pipeline 0	

Total database accesses: 1, total allocated memory: 176

## 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.

## Example 382. Distinct

### Query

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

### Query Plan

Compiler CYPHER 4.4

Planner COST

Runtime PIPELINED

Runtime version 4.4

Operator	Details	Estimated Rows	Rows	DB Hits	Memory (Bytes)
+ProduceResults	l Fused in Pipeline 0	14	6	0	
+Distinct	l Fused in Pipeline 0	14	6	0	224
+Filter	p:Person Fused in Pipeline 0	15	15	15	
+Expand(All)	(l)<-[anon_0:WORKS_IN]-(p) Fused in Pipeline 0	15	15	16	
+Filter	l:Location Fused in Pipeline 0	10	10	0	
+AllNodesScan	l 5/0   0.628   Fused in Pipeline 0	35	35	36	112

Total database accesses: 67, total allocated memory: 304

## 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.

## Example 383. OrderedDistinct

### Query

```
MATCH (p:Person)
WHERE p.name STARTS WITH 'P'
RETURN DISTINCT p.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) | Ordered by |
| Other           |        |         |                |                        |           |           |
+-----+-----+-----+-----+-----+-----+-----+
| +ProduceResults  | `p.name` |
| 0 | 2 | 0 | | 0/0 | 0.071 | p.name ASC |
| In Pipeline 0 |
| |
+-----+-----+-----+-----+-----+-----+-----+
| +OrderedDistinct | cache[p.name] AS `p.name` |
| 0 | 2 | 0 | 32 | 0/0 | 1.923 | 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 | 0.239 |
| p.name ASC | In Pipeline 0 |
+-----+-----+-----+-----+-----+-----+-----+
Total database accesses: 3, total allocated memory: 176
```

## Filter

The **Filter** operator filters each row coming from the child operator, only passing through rows that evaluate the predicates to **true**.

## Example 384. Filter

### Query

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

### 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                                           | 14
| 0 | 0 | | | | Fused in Pipeline 0 |
| | | | | | |
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
| +Filter          | cache[p.name] =~ $autostring_0           | 14
| 0 | 0 | | | | Fused in Pipeline 0 |
| | | | | | |
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
| +NodeIndexScan | BTREE INDEX p:Person(name) WHERE name IS NOT NULL, cache[p.name] | 14
| 14 | 15 | 112 | 0/1 | 0.373 | Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
```

Total database accesses: 15, total allocated memory: 176

## Limit

The `Limit` operator returns the first `n` rows from the incoming input.

## Example 385. Limit

### Query

```
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	0.320	Fused in Pipeline 0				

Total database accesses: 5, total allocated memory: 176

## Skip

The **Skip** operator skips **n** rows from the incoming rows.

## Example 386. Skip

### Query

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

### 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
+ProduceResults	p	13	13	0		
2/0	0.698   p.id ASC   In Pipeline 1					
+Skip	\$autoint_0	13	13	0		32
0/0	2.206   p.id ASC   In Pipeline 1					
+Sort	'p.id' ASC	14	14	0		1800
0/0	0.708   p.id ASC   In Pipeline 1					
+Projection	p.id AS 'p.id'	14	14	0		
	Fused in Pipeline 0					
+Filter	p:Person	14	14	0		
	Fused in Pipeline 0					
+AllNodesScan	p	35	35	36		112
2/0	1.153   Fused in Pipeline 0					

Total database accesses: 36, total allocated memory: 1912

## 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.

## Example 387. Sort

### Query

```
MATCH (p:Person)
RETURN p
ORDER BY p.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) | Ordered by | Other |         |
+-----+-----+-----+-----+-----+-----+
| +ProduceResults  | p                                     | 14 | 14 | 0 | |
2/0 | 1.224 | p.name ASC | In Pipeline 1 | |
| |
+-----+-----+-----+-----+-----+-----+
| +Sort            | `p.name` ASC                          | 14 | 14 | 0 | 2592 |
0/0 | 0.922 | p.name ASC | In Pipeline 1 | |
| |
+-----+-----+-----+-----+-----+-----+
| +Projection      | p.name AS `p.name`                    | 14 | 14 | 14 | |
| |                               | Fused in Pipeline 0 |
| |
+-----+-----+-----+-----+-----+-----+
| +Filter          | p:Person                               | 14 | 14 | 0 | |
| |                               | Fused in Pipeline 0 |
| |
+-----+-----+-----+-----+-----+-----+
| +AllNodesScan    | p                                       | 35 | 35 | 36 | 112 |
2/0 | 0.990 | | Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
```

Total database accesses: 50, total allocated memory: 2672

## 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.



## Example 388. PartialSort

### 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

```
+-----+
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
+-----+
| Operator          | Details
| Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Ordered by
| Other          |
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
| +ProduceResults  | p
|           0 | 2 | 0 |           | 2/0 | 0.550 | p.name ASC,
p.age ASC | In Pipeline 1 |
| |
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
| +PartialSort     | `p.name` ASC, `p.age` ASC
|           0 | 2 | 0 | 3096 | 0/0 | 0.535 | p.name ASC,
p.age ASC | In Pipeline 1 |
| |
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
| +Projection      | cache[p.name] AS `p.name`, p.age AS `p.age`
|           0 | 2 | 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 | 1.068
| p.name ASC          | Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
```

Total database accesses: 3, total allocated memory: 3160

## 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.

## Example 389. Top

### Query

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

### 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)	Ordered by	Other
+ProduceResults	p	2	2	0			0.093	p.name ASC	In Pipeline 1
+Top	`p.name` ASC LIMIT 2	2	2	0	2584		0.638	p.name ASC	In Pipeline 1
+Projection	p.name AS `p.name`	14	14	14					Fused in Pipeline 0
+Filter	p:Person	14	14	0					Fused in Pipeline 0
+AllNodesScan	p	35	35	36	112		0.173		Fused in Pipeline 0

Total database accesses: 50, total allocated memory: 2664

## 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.



## Example 391. Union

### Query

```
MATCH (p:Location)
RETURN p.name
UNION ALL
MATCH (p:Country)
RETURN p.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
+ProduceResults	`p.name` Fused in Pipeline 2	20	11	0		
+Union		20	11	0	2224	
+Projection	`p.name` Fused in Pipeline 1	10	1	0		
+Projection	p.name AS `p.name` Fused in Pipeline 1	10	1	1		
+Filter	p:Country Fused in Pipeline 1	10	1	0		
+AllNodesScan	p Fused in Pipeline 1	35	35	36	112	
+Projection	`p.name` Fused in Pipeline 0	10	10	0		
+Projection	p.name AS `p.name` Fused in Pipeline 0	10	10	10		
+Filter	p:Location Fused in Pipeline 0	10	10	0		
+AllNodesScan	p Fused in Pipeline 0	35	35	36	112	

Total database accesses: 83, total allocated memory: 2424

# Unwind

The **Unwind** operator returns one row per item in a list.

## Example 392. Unwind

### Query

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

### Query Plan

Compiler CYPHER 4.4

Planner COST

Runtime PIPELINED

Runtime version 4.4

Operator	Details	Estimated Rows	Rows	DB Hits	Page
+ProduceResults	value	10	5	0	
	Fused in Pipeline 0				
+Unwind	range(\$autoint_0, \$autoint_1) AS value	10	5	0	
0/0	0.000 Fused in Pipeline 0				

Total database accesses: 0, total allocated memory: 176

## Exhaustive Limit

The **ExhaustiveLimit** operator is just like a normal **Limit** but will always exhaust the input. Used when combining **LIMIT** and updates

## Example 393. ExhaustiveLimit

### Query

```
MATCH (p:Person)
SET p.seen = true
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							
+ExhaustiveLimit	3	3	3	0	32			
	Fused in Pipeline 0							
+SetProperty	p.seen = true	14	14	28				
	Fused in Pipeline 0							
+Filter	p:Person	14	14	0				
	Fused in Pipeline 0							
+AllNodesScan	p	35	35	36	112	3/0	1.062	Fused in Pipeline 0

Total database accesses: 64, total allocated memory: 192

## 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**.

## Example 394. Optional

### 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
+ProduceResults	p, q	1	1
0	2/0   0.097   In Pipeline 2		
+Apply		1	1
0	0/0   0.012		
+Optional	p	1	1
0	768   0/0   0.339   In Pipeline 2		
+NodeIndexSeek	BTREE INDEX q:Person(name) WHERE name = \$autostring_1	1	0
1	2152   1/0   0.483   In Pipeline 1		
+NodeIndexSeek	BTREE INDEX p:Person(name) WHERE name = \$autostring_0	1	1
2	112   0/1   0.188   In Pipeline 0		

Total database accesses: 3, total allocated memory: 3000

## Project Endpoints

The `ProjectEndpoints` operator projects the start and end node of a relationship.

## Example 395. ProjectEndpoints

### Query

```
CREATE (n)-[p:KNOWS]->(m)
WITH p AS r
MATCH (u)-[r]->(v)
RETURN u, v
```

### Query Plan

Compiler CYPHER 4.4

Planner COST

Runtime PIPELINED

Runtime version 4.4

Operator (Bytes)	Details Page Cache Hits/Misses	Time (ms)   Other	Estimated Rows	Rows	DB Hits	Memory
+ProduceResults	u, v	Fused in Pipeline 1	1	1	0	
+Apply			1	1	0	
+ProjectEndpoints	(u)-[r*]->(v)	Fused in Pipeline 1	1	1	0	
+Argument	r	0/0   0.160   Fused in Pipeline 1	1	1	0	
+Projection	p AS r	Fused in Pipeline 0	1	1	0	
+Create	(n), (m), (n)-[p:KNOWS]->(m)	0/0   0.000   Fused in Pipeline 0	1	1	4	

Total database accesses: 4, total allocated memory: 4280

## Projection

For each incoming row, the **Projection** operator evaluates a set of expressions and produces a row with the results of the expressions.



## Example 396. Projection

### Query

```
RETURN 'hello' AS greeting
```

### Query Plan

Compiler CYPHER 4.4

Planner COST

Runtime PIPELINED

Runtime version 4.4

```
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+
| Operator      | Details                | Estimated Rows | Rows | DB Hits | Page Cache
Hits/Misses | Time (ms) | Other          |      |      |      |
+-----+-----+-----+-----+-----+
| +ProduceResults | greeting                |          1 | 1 | 0 |
|           | Fused in Pipeline 0 |
| |           +-----+-----+-----+-----+
| |           +-----+-----+
| +Projection    | $autostring_0 AS greeting |          1 | 1 | 0 |
0/0 | 0.000 | Fused in Pipeline 0 |
+-----+-----+-----+-----+
+-----+-----+-----+-----+

```

Total database accesses: 0, total allocated memory: 176

## Shortest path

The **ShortestPath** operator finds one or all shortest paths between two previously matches node variables.

## Example 397. ShortestPath

### Query

```
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

Operator	Details	Estimated Rows
+ProduceResults	p	1
1	0   1/0   0.169   In Pipeline 1	
+ShortestPath	p = (andy)-[anon_0*]-(mattias)	1
1	1   3176     In Pipeline 1	
+MultiNodeIndexSeek	BTREE INDEX andy:Person(name) WHERE name = \$autostring_0, BTREE INDEX mattias:Person(name) WHERE name = \$autostring_1	1
1	4   112   1/1   0.277   In Pipeline 0	

Total database accesses: 5, total allocated memory: 3240

## Empty Row

The `EmptyRow` operator returns a single row with no columns.

## Example 398. EmptyRow

### Query

```
CYPHER runtime=slotted
FOREACH (value IN [1,2,3] | MERGE (:Person {age: value}))
```

### 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		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

## Procedure Call

The `ProcedureCall` operator indicates an invocation to a procedure.

## Example 399. ProcedureCall

### Query

```
CALL db.labels() YIELD label
RETURN *
ORDER BY label
```

### 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) | Ordered by | Other |
+-----+-----+-----+-----+-----+
| +ProduceResults | label                                 | 10 | 4 | 0 |
| |                | 0/0 | 0.062 | label ASC | In Pipeline 1 |
| |                +-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
| +Sort          | label ASC                             | 10 | 4 | 0 |
1064 |                | 0/0 | 0.177 | label ASC | In Pipeline 1 |
| |                +-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
| +ProcedureCall | db.labels() :: (label :: STRING?) | 10 | 4 |  |
| |                |                | Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
```

Total database accesses: ?, total allocated memory: 1128

## 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.

## Example 400. CacheProperties

### Query

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

### Query Plan

Compiler CYPHER 4.4

Planner COST

Runtime PIPELINED

Runtime version 4.4

Operator	Details	Estimated Rows	Rows	DB Hits
+ProduceResults	location, name   Fused in Pipeline 0	15	15	0
+Projection	cache[l.name] AS location, p.name AS name   Fused in Pipeline 0	15	15	30
+Filter	p:Person   Fused in Pipeline 0	15	15	15
+Expand(All)	(l)-[:WORKS_IN]-(p)   Fused in Pipeline 0	15	15	16
+CacheProperties	cache[l.name]   Fused in Pipeline 0	10	10	10
+Filter	l:Location   Fused in Pipeline 0	10	10	0
+AllNodesScan	l 112   4/0   0.410   Fused in Pipeline 0	35	35	36

Total database accesses: 107, total allocated memory: 192

## Create (nodes and relationships)

The **Create** operator is used to create nodes and relationships.

## Example 401. Create

### Query

```
CREATE
  (max:Person {name: 'Max'}),
  (chris:Person {name: 'Chris'})
CREATE (max)-[:FRIENDS_WITH]->(chris)
```

### Query Plan

Compiler CYPHER 4.4

Planner COST

Runtime PIPELINED

Runtime version 4.4

Operator	Details
Estimated Rows	Rows   DB Hits   Page Cache Hits/Misses   Time (ms)   Other
+ProduceResults	
1   0   0	Fused in Pipeline 0
+EmptyResult	
1   0   0	Fused in Pipeline 0
+Create	(max:Person {name: \$autostring_0}), (chris:Person {name: \$autostring_1}), (max)-[anon_0:FRIENDS_WITH]->(chris)
1   1   7	0/0   0.000   Fused in Pipeline 0

Total database accesses: 7, total allocated memory: 176

## Delete (nodes and relationships)

The **Delete** operator is used to delete a node or a relationship.

## Example 402. Delete

### Query

```
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 | 0.215 | 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 | 0.442 | Fused in Pipeline 0 |
| |
+-----+
+-----+
```

Total database accesses: 10, total allocated memory: 3272

## Detach Delete

The `DetachDelete` operator is used in all queries containing the `DETACH DELETE` clause, when deleting nodes and their relationships.

## Example 403. DetachDelete

### Query

```
MATCH (p:Person)
DETACH DELETE p
```

### 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		14	0	0				
	Fused in Pipeline 0							
+EmptyResult		14	0	0				
	Fused in Pipeline 0							
+DetachDelete	p	14	14	31				
	Fused in Pipeline 0							
+Filter	p:Person	14	14	0				
	Fused in Pipeline 0							
+AllNodesScan	p	35	35	36		112	19/0	1.332
	Fused in Pipeline 0							

Total database accesses: 67, total allocated memory: 192

## Set Labels

The `SetLabels` operator is used when setting labels on a node.



## Example 404. SetLabels

### Query

```
MATCH (n)
SET n: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		35	0	0				
	Fused in Pipeline 0							
+EmptyResult		35	0	0				
	Fused in Pipeline 0							
+SetLabels	n:Person	35	35	22				
	Fused in Pipeline 0							
+AllNodesScan	n	35	35	36				
3/0	1.188				112			Fused in Pipeline 0

Total database accesses: 58, total allocated memory: 176

## Remove Labels

The `RemoveLabels` operator is used when deleting labels from a node.

## Example 405. RemoveLabels

### Query

```
MATCH (n)
REMOVE n: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		35	0	0				
	Fused in Pipeline 0							
+EmptyResult		35	0	0				
	Fused in Pipeline 0							
+RemoveLabels	n:Person	35	35	15				
	Fused in Pipeline 0							
+AllNodesScan	n	35	35	36				
	Fused in Pipeline 0							

Total database accesses: 51, total allocated memory: 176

## Set Node Properties From Map

The `SetNodePropertiesFromMap` operator is used when setting properties from a map on a node.

## Example 406. SetNodePropertiesFromMap

### Query

```
MATCH (n)
SET n = {weekday: 'Monday', meal: 'Lunch'}
```

### 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		35
0	0         Fused in Pipeline 0	
+EmptyResult		35
0	0         Fused in Pipeline 0	
+SetNodePropertiesFromMap	n = {weekday: \$autostring_0, meal: \$autostring_1}	35
35	105         Fused in Pipeline 0	
+AllNodesScan	n	35
35	36   112   5/0   3.873   Fused in Pipeline 0	

Total database accesses: 141, total allocated memory: 176

## Set Relationship Properties From Map

The `SetRelationshipPropertiesFromMap` operator is used when setting properties from a map on a relationship.

## Example 407. SetRelationshipPropertiesFromMap

### Query

```
MATCH (n)-[r]->(m)
SET r = {weight: 5, unit: 'kg'}
```

### Query Plan

Compiler CYPHER 4.4

Planner COST

Runtime PIPELINED

Runtime version 4.4

Operator	Rows	DB Hits	Memory (Bytes)	Details	Page Cache Hits/Misses	Time (ms)	Other	Estimated Rows
+ProduceResults	0	0					Fused in Pipeline 0	18
+EmptyResult	0	0					Fused in Pipeline 0	18
+SetRelationshipPropertiesFromMap	18	54		r = {weight: \$autoint_0, unit: \$autostring_1}			Fused in Pipeline 0	18
+Expand(All)	18	36		(m)<-[r]-(n)			Fused in Pipeline 0	18
+AllNodesScan	35	36	112	m	6/0	4.009	Fused in Pipeline 0	35

Total database accesses: 126, total allocated memory: 192

## Set Property

The **SetProperty** operator is used when setting a property on a node or relationship.

## Example 408. SetProperty

### Query

```
MATCH (n)
SET n.checked = true
```

### 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		35	0	0				
	Fused in Pipeline 0							
+EmptyResult		35	0	0				
	Fused in Pipeline 0							
+SetProperty	n.checked = true	35	35	70				
	Fused in Pipeline 0							
+AllNodesScan	n	35	35	36				
3/0	0.529							Fused in Pipeline 0

Total database accesses: 106, total allocated memory: 176

## 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.

### Example 409. CreateUniqueConstraint

#### 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

Operator	Details
+CreateConstraint	CONSTRAINT uniqueness FOR (c:Country) REQUIRE (c.name) IS UNIQUE

Total database accesses: ?

## 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.

### Example 410. DropUniqueConstraint

#### Query

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

#### Query Plan

Compiler CYPHER 4.4

Planner ADMINISTRATION

Runtime SCHEMA

Runtime version 4.4

Operator	Details
+DropConstraint	CONSTRAINT ON (c:Country) ASSERT (c.name) IS UNIQUE

Total database accesses: ?

## Do Nothing If Exists (constraint)

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.

Example 411. `DoNothingIfExists(CONSTRAINT)`

### Query

```
CREATE CONSTRAINT uniqueness IF NOT EXISTS
FOR (c:Country) REQUIRE c.name IS UNIQUE
```

### Query Plan

Compiler CYPHER 4.4

Planner ADMINISTRATION

Runtime SCHEMA

Runtime version 4.4

Operator	Details
+CreateConstraint	CONSTRAINT uniqueness FOR (c:Country) REQUIRE (c.name) IS UNIQUE
+DoNothingIfExists(CONSTRAINT)	CONSTRAINT uniqueness FOR (c:Country) REQUIRE (c.name) IS UNIQUE

Total database accesses: ?

## Create Node Property Existence Constraint

The `CreateNodePropertyExistenceConstraint` 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.

## Example 412. CreateNodePropertyExistenceConstraint

### Query

```
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

```
+-----+-----+
| Operator          | Details                                     |
+-----+-----+
| +CreateConstraint | CONSTRAINT existence FOR (p:Person) REQUIRE (p.name) IS NOT NULL |
+-----+-----+
```

Total database accesses: ?

## Drop Node Property Existence Constraint Deprecated

The `DropNodePropertyExistenceConstraint` operator removes an existence constraint from a property for all nodes having a certain label. This will only appear in Enterprise Edition.

## Example 413. DropNodePropertyExistenceConstraint

### Query

```
DROP CONSTRAINT ON (p:Person) ASSERT exists(p.name)
```

### Query Plan

Compiler CYPHER 4.4

Planner ADMINISTRATION

Runtime SCHEMA

Runtime version 4.4

```
+-----+-----+
| Operator          | Details                                     |
+-----+-----+
| +DropConstraint  | CONSTRAINT ON (p:Person) ASSERT exists(p.name) |
+-----+-----+
```

Total database accesses: ?

## 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.

#### Example 414. CreateNodeKeyConstraint

##### Query

```
CREATE CONSTRAINT node_key
FOR (e:Employee) REQUIRE (e.firstname, e.surname) IS NODE KEY
```

##### Query Plan

```
Compiler CYPHER 4.4
Planner ADMINISTRATION
Runtime SCHEMA
Runtime version 4.4
+-----+
+-----+
| Operator          | Details
+-----+
+-----+
| +CreateConstraint | CONSTRAINT node_key FOR (e:Employee) REQUIRE (e.firstname, e.surname) IS NODE
KEY |
+-----+
+-----+
Total database accesses: ?
```

## 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.

#### Example 415. DropNodeKeyConstraint

##### Query

```
DROP CONSTRAINT ON (e:Employee) ASSERT (e.firstname, e.surname) IS NODE KEY
```

##### Query Plan

```
Compiler CYPHER 4.4
Planner ADMINISTRATION
Runtime SCHEMA
Runtime version 4.4
+-----+
+-----+
| Operator          | Details
+-----+
+-----+
| +DropConstraint  | CONSTRAINT ON (e:Employee) ASSERT (e.firstname, e.surname) IS NODE KEY |
+-----+
+-----+
Total database accesses: ?
```

## 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.

Example 416. `CreateRelationshipPropertyExistenceConstraint`

### Query

```
CREATE CONSTRAINT existence
FOR ()-[1:LIKED]-() REQUIRE 1.when IS NOT NULL
```

### Query Plan

Compiler CYPHER 4.4

Planner ADMINISTRATION

Runtime SCHEMA

Runtime version 4.4

Operator	Details
+CreateConstraint	CONSTRAINT existence FOR ()-[1:LIKED]-() REQUIRE (1.when) IS NOT NULL

Total database accesses: ?

## Drop Relationship Property Existence Constraint Deprecated

The `DropRelationshipPropertyExistenceConstraint` operator removes an existence constraint from a property for all relationships of a certain type. This will only appear in Enterprise Edition.

### Example 417. DropRelationshipPropertyExistenceConstraint

#### Query

```
DROP CONSTRAINT ON ()-[1:LIKED]-() ASSERT exists(1.when)
```

#### Query Plan

Compiler CYPHER 4.4

Planner ADMINISTRATION

Runtime SCHEMA

Runtime version 4.4

Operator	Details
+DropConstraint	CONSTRAINT ON ()-[1:LIKED]-() ASSERT exists(1.when)

Total database accesses: ?

## Drop Constraint

The **DropConstraint** operator removes a constraint using the name of the constraint, no matter the type.

### Example 418. DropConstraint

#### 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: ?

## Show Constraints

The **ShowConstraints** operator lists constraints. It may include filtering on constraint type and can have either default or full output.

## Example 419. ShowConstraints

### Query

```
SHOW CONSTRAINTS
```

### 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  | id, name, type, entityType, labelsOrTypes, properties, ownedIndexId |
10 | 3 | 0 | 0/0 |
| |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| +ShowConstraints | allConstraints, defaultColumns |
10 | 3 | 1 | 0/0 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

```
Total database accesses: 1, total allocated memory: 64
```

## 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.

## Example 420. CreateIndex

### Query

```
CREATE INDEX my_index
FOR (c:Country) ON (c.name)
```

### Query Plan

Compiler CYPHER 4.4

Planner ADMINISTRATION

Runtime SCHEMA

Runtime version 4.4

Operator	Details
+CreateIndex	BTREE INDEX my_index FOR (:Country) ON (name)

Total database accesses: ?

## Do Nothing If Exists (index)

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.

## Example 421. DoNothingIfExists(INDEX)

### Query

```
CREATE INDEX my_index IF NOT EXISTS
FOR ()-[k:KNOWS]-() ON (k.since)
```

### Query Plan

Compiler CYPHER 4.4

Planner ADMINISTRATION

Runtime SCHEMA

Runtime version 4.4

Operator	Details
+CreateIndex	BTREE INDEX my_index FOR ()-[k:KNOWS]-() ON (since)
+DoNothingIfExists(INDEX)	BTREE INDEX my_index FOR ()-[k:KNOWS]-() ON (since)

Total database accesses: ?

## Drop Index by schema Deprecated

The `DropIndex` operator removes an index from a property for all nodes having a certain label.

### Example 422. DropIndex

The following query will drop an index on the `name` property of nodes with the `Country` label.

#### Query

```
DROP INDEX ON :Country(name)
```

#### Query Plan

Compiler CYPHER 4.4

Planner ADMINISTRATION

Runtime SCHEMA

Runtime version 4.4

```
+-----+-----+
| Operator | Details |
+-----+-----+
| +DropIndex | BTREE INDEX FOR (:Country) ON (name) |
+-----+-----+
```

Total database accesses: ?

## Drop Index

The `DropIndex` operator removes an index using the name of the index.

### Example 423. DropIndex

#### Query

```
DROP INDEX name
```

#### Query Plan

Compiler CYPHER 4.4

Planner ADMINISTRATION

Runtime SCHEMA

Runtime version 4.4

```
+-----+-----+
| Operator | Details |
+-----+-----+
| +DropIndex | INDEX name |
+-----+-----+
```

Total database accesses: ?

## Show Indexes

The `ShowIndexes` operator lists indexes. It may include filtering on index type and can have either default or full output.

### Example 424. ShowIndexes

#### Query

```
SHOW INDEXES
```

#### 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 | id, name, state, populationPercent, uniqueness, type, entityType, labelsOrTypes,
properties, |      10 | 7 | 0 | 0/0 |
| |
| | indexProvider
| |
| |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
| +ShowIndexes   | allIndexes, defaultColumns
|      10 | 7 | 1 | 0/0 |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
```

```
Total database accesses: 1, total allocated memory: 64
```

## Show 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.

## Example 425. ShowFunctions

### Query

```
SHOW FUNCTIONS
```

### 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, category, description           |              10 | 143 | |
0 |              0/0 |
| |              +-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| +ShowFunctions  | allFunctions, functionsForUser(all), defaultColumns |              10 | 143 | |
0 |              0/0 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

```
Total database accesses: 0, total allocated memory: 64
```

## Show 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.



## Example 426. ShowProcedures

### 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
```

## Show Transactions

The `ShowTransactions` operator lists transactions. It may include filtering on given ids and can have either default or full output.

## Example 427. ShowTransactions

### Query

```
SHOW TRANSACTIONS
```

### 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 | database, transactionId, currentQueryId, connectionId, clientAddress, username,
currentQuery, |      10 | 1 | 0 | 0/0 |
| |
| | | |
| | | |
+-----+-----+-----+-----+
| +ShowTransactions | defaultColumns, allTransactions
|      10 | 1 | 0 | 0/0 |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
```

```
Total database accesses: 0, total allocated memory: 64
```

## Terminate Transactions

The `TerminateTransactions` operator terminates transactions by ID.

## Example 428. TerminateTransactions

### Query

```
TERMINATE TRANSACTIONS 'database-transaction-123'
```

### 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   | transactionId, username, message     |          10 |    1 |    0 | |
| 0/0 |             |                                         |                |      |         |
| |                 |                                         |                |      |         |
+-----+-----+-----+-----+-----+
| +TerminateTransactions | transactions: database-transaction-123 |          10 |    1 |    0 |
| 0/0 |             |                                         |                |      |         |
+-----+-----+-----+-----+-----+
+-----+
```

Total database accesses: 0, total allocated memory: 64

## Shortest path planning

*Shortest path finding in Cypher and how it is planned.*

Planning shortest paths in Cypher can lead to different query plans depending on the predicates that need to be evaluated. Internally, Neo4j will use a fast bidirectional breadth-first search algorithm if the predicates can be evaluated whilst searching for the path. Therefore, this fast algorithm will always be certain to return the right answer when there are universal predicates on the path; for example, when searching for the shortest path where all nodes have the **Person** label, or where there are no nodes with a **name** property.

If the predicates need to inspect the whole path before deciding on whether it is valid or not, this fast algorithm cannot be relied on to find the shortest path, and Neo4j may have to resort to using a slower exhaustive depth-first search algorithm to find the path. This means that query plans for shortest path queries with non-universal predicates will include a fallback to running the exhaustive search to find the path should the fast algorithm not succeed. For example, depending on the data, an answer to a shortest path query with existential predicates — such as the requirement that at least one node contains the property **name='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](#).

## Shortest path — fast algorithm



Example 430. Consider using the exhaustive search as a fallback

### Query

```
MATCH
  (KevinB:Person {name: 'Kevin Bacon'}),
  (Al:Person {name: 'Al Pacino'}),
  p = shortestPath((KevinB)-[*]-(Al))
WHERE length(p) > 1
RETURN p
```

This query, in contrast with the one above, needs to check that the whole path follows the predicate before we know if it is valid or not, and so the query plan will also include the fallback to the slower exhaustive search algorithm.

### Query plan

```
Compiler CYPHER 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                                           |
1 | 1 | 0 | | | | | Fused in Pipeline 6 |
| | |
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
| +AntiConditionalApply | 41456 | 0/0 | 0.496 | Fused in Pipeline 6 |
1 | 1 | 0 | 41456 | | | |
| | \
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
| | +Top            | anon_1 ASC LIMIT 1                         |
2 | 0 | 0 | 4280 | | 0/0 | 0.000 | In Pipeline 5 |
| | |
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
| | +Projection     | length(p) AS anon_1                       |
7966 | 0 | 0 | | | | | Fused in Pipeline 4 |
| | |
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
| | +Filter         | length(p) > $autoint_2                    |
7966 | 0 | 0 | | | | | Fused in Pipeline 4 |
| | |
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
| | +Projection     | (KevinB)-[anon_0*]-(Al) AS p              |
26554 | 0 | 0 | | | | | Fused in Pipeline 4 |
| | |
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
| | +VarLengthExpand(Into) | (KevinB)-[anon_0*]-(Al)                  |
26554 | 0 | 0 | | | | | Fused in Pipeline 4 |
| | |
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
| | +Argument       | KevinB, Al                                 |
2 | 0 | 0 | 0 | 0/0 | 0.000 | Fused in Pipeline 4 |
| | |
+-----+-----+-----+-----+-----+-----+-----+-----+
```



### Example 431. Prevent the exhaustive search from being used as a fallback

#### Query

```
MATCH
  (KevinB:Person {name: 'Kevin Bacon'}),
  (Al:Person {name: 'Al Pacino'}),
  p = shortestPath((KevinB)-[*]-(Al))
WITH p
WHERE length(p) > 1
RETURN p
```

This query, just like the one above, needs to check that the whole path follows the predicate before we know if it is valid or not. However, the inclusion of the **WITH** clause means that the query plan will not include the fallback to the slower exhaustive search algorithm. Instead, any paths found by the fast algorithm will subsequently be filtered, which may result in no answers being returned.

#### Query plan

Compiler CYPHER 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	1
1	0   1/0   0.118   In Pipeline 1	
+Filter	length(p) > \$autoint_2	1
1	0   0/0   0.063   In Pipeline 1	
+ShortestPath	p = (KevinB)-[anon_0*]-(Al)	2
1	1   3176     In Pipeline 1	
+MultiNodeIndexSeek	BTREE INDEX KevinB:Person(name) WHERE name = \$autostring_0,	2
1	4   112   2/0   0.226   In Pipeline 0	
	BTREE INDEX Al:Person(name) WHERE name = \$autostring_1	

Total database accesses: 5, total allocated memory: 3240



# Administration

Cypher contains a number of administrative commands which allows for efficient and sophisticated database management, alias management, server management, and role-based access-control.

More information about each of these topics can be found in the following sections:

- [Database management](#)
- [Database alias management](#)
- [Access control](#)

## 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 502. Special characters in syntax summaries

Character	Meaning	Example
	Used to indicate alternative parts of a command (i.e. <i>or</i> ). Needs to be part of a grouping.	If the syntax needs to specify either a name or <i>*</i> , this can be indicated with <i>*   name</i> .
{ and }	Used to group parts of the command. Commonly found together with  .	In order to use the <i>or</i> in the syntax summary, it needs to be in a group: <i>{*   name}</i> .
[ and ]	Used to indicate an optional part of the command. It also groups alternatives together, when there can be either of the alternatives or nothing.	If a keyword in the syntax can either be in singular or plural, we can indicate that the <i>S</i> is optional with <i>GRAPH[S]</i> .
...	Repeated pattern. Related to the command part immediately before this is repeated.	A comma separated list of names would be <i>name[, ...]</i> .
"	When a special character is part of the syntax itself, we surround it with " to indicate this.	To include { in the syntax use <i>"{ *   name }"</i> . In this case, you will get either <i>* }</i> or <i>{ name }</i> .

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 in granting the **READ** privilege:

```
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.

```
CREATE CONSTRAINT [constraint_name] [IF NOT EXISTS]
FOR (n:LabelName)
REQUIRE n.propertyName IS NOT NULL
```

## Database management

*This section explains how to use Cypher to manage databases in Neo4j DBMS: 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:



More details about the syntax descriptions can be found [here](#).

Table 503. Database management command syntax

Command	Syntax
<b>SHOW DATABASE</b>	<pre>SHOW { DATABASE name   DATABASES   DEFAULT DATABASE   HOME DATABASE } [WHERE expression]</pre> <pre>SHOW { DATABASE name   DATABASES   DEFAULT DATABASE   HOME DATABASE } YIELD { *   field[, ...] } [ORDER BY field[, ...]] [SKIP n] [LIMIT n] [WHERE expression] [RETURN field[, ...] [ORDER BY field[, ...]] [SKIP n] [LIMIT n]]</pre>

Command	Syntax
CREATE DATABASE	<pre>CREATE DATABASE name [IF NOT EXISTS] [OPTIONS "{" option: value[, ...] "}"] [WAIT [n [SEC[OND[S]]]] NOWAIT</pre> <pre>CREATE OR REPLACE DATABASE name [WAIT [n [SEC[OND[S]]]] NOWAIT</pre>
ALTER DATABASE	<pre>ALTER DATABASE name [IF EXISTS] SET ACCESS {READ ONLY   READ WRITE}</pre>
STOP DATABASE	<pre>STOP DATABASE name [WAIT [n [SEC[OND[S]]]] NOWAIT</pre>
START DATABASE	<pre>START DATABASE name [WAIT [n [SEC[OND[S]]]] NOWAIT</pre>
DROP DATABASE	<pre>DROP DATABASE name [IF EXISTS] [{DUMP DESTROY} [DATA]] [WAIT [n [SEC[OND[S]]]] NOWAIT</pre>

## Listing databases



There are four different commands for listing databases:

- Listing all databases.
- Listing a particular database.
- Listing the default database.
- Listing the home database.

These commands return the following columns:

Table 504. Listing databases output

Column	Description
name	The name of the database. <a href="#">Default output</a>
aliases	The names of any aliases the database may have. <a href="#">Default output</a>
access	The database access mode, either <code>read-write</code> or <code>read-only</code> . <a href="#">Default output</a>
databaseID	The database unique ID.

Column	Description
<code>serverID</code>	The server instance ID.
<code>address</code>	Instance address in a clustered DBMS. The default for a standalone database is <code>neo4j://localhost:7687</code> . <a href="#">Default output</a>
<code>role</code>	The current role of the database ( <code>standalone</code> , <code>leader</code> , <code>follower</code> , <code>read_replica</code> , <code>unknown</code> ). <a href="#">Default output</a>
<code>requestedStatus</code>	The expected status of the database. <a href="#">Default output</a>
<code>currentStatus</code>	The actual status of the database. <a href="#">Default output</a>
<code>error</code>	An error message explaining why the database is not in the correct state. <a href="#">Default output</a>
<code>default</code>	Show if this is the default database for the DBMS. <a href="#">Default output</a> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;">  Not returned by <code>SHOW HOME DATABASE</code> or <code>SHOW DEFAULT DATABASE</code>. </div>
<code>home</code>	Shown if this is the home database for the current user. <a href="#">Default output</a> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;">  Not returned by <code>SHOW HOME DATABASE</code> or <code>SHOW DEFAULT DATABASE</code>. </div>
<code>lastCommittedTxn</code>	The ID of the last transaction received.
<code>replicationLag</code>	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 <code>0</code> .

## Example 432. SHOW DATABASES

A summary of all available databases can be displayed using the command `SHOW DATABASES`.

Query

```
SHOW DATABASES
```

Table 505. Result

name	aliases	access	address	role	requested Status	currentStatus	error	default	home
"movies"	["films", "motion pictures"]	"read-write"	"localhost:7687"	"standalone"	"online"	"online"	" "	false	false
"neo4j"	[]	"read-write"	"localhost:7687"	"standalone"	"online"	"online"	" "	true	true
"system"	[]	"read-write"	"localhost:7687"	"standalone"	"online"	"online"	" "	false	false

Rows: 3



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.

### Example 433. SHOW DATABASES

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 506. Result

name	aliases	access	databaseID	serverID	addresses	role	requestedStatus	currentStatus	error	default	home	lastCommittedTxn	replicationLag
"movies"	["films", "motion pictures"]	"read-write"	"4917FA22626DD62C8955FDE7D0CAE0C971E990C4DD1DDB3AECB8CE4C6798B484"	"8af5d882-d4ce-4ce7-a3a5-7e550967f06b"	"localhost:7687"	"standalone"	"online"	"online"	"	false	false	3	0

Rows: 1

### Example 434. SHOW DATABASES

The number of databases can be seen using a `count()` aggregation with `YIELD` and `RETURN`.

#### Query

```
SHOW DATABASES YIELD *  
RETURN count(*) AS count
```

#### Table 507. Result

count
3

Rows: 1

### Example 435. SHOW DEFAULT DATABASE

The default database can be seen using the command `SHOW DEFAULT DATABASE`.

Query

```
SHOW DEFAULT DATABASE
```

Table 508. Result

name	aliases	access	address	role	requestedStatus	currentStatus	error
"neo4j"	[]	"read-write"	"localhost:7687"	"standalone"	"online"	"online"	" "

Rows: 1

### Example 436. SHOW HOME DATABASE

The home database for the current user can be seen using the command `SHOW HOME DATABASE`.

Query

```
SHOW HOME DATABASE
```

Table 509. Result

name	aliases	access	address	role	requestedStatus	currentStatus	error
"neo4j"	[]	"read-write"	"localhost:7687"	"standalone"	"online"	"online"	" "

Rows: 1

## Example 437. SHOW DATABASES

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 510. Result

name	currentStatus	requestedStatus
"movies"	"online"	"online"
"neo4j"	"online"	"online"
"system"	"online"	"online"

Rows: 3



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`.

## Creating databases Enterprise edition

Databases can be created using `CREATE DATABASE`.



## Example 438. CREATE DATABASE

### Query

```
CREATE DATABASE customers
```

### Result

```
System updates: 1  
Rows: 0
```



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.

### Example 439. SHOW DATABASES

When a database has been created, it will show up in the listing provided by the command `SHOW DATABASES`.

Query

```
SHOW DATABASES
```

Table 511. Result

name	aliases	access	address	role	requested Status	currentStatus	error	default	home
"customers"	[]	"read-write"	"localhost:7687"	"standalone"	"online"	"online"	""	false	false
"movies"	["films", "motion pictures"]	"read-write"	"localhost:7687"	"standalone"	"online"	"online"	""	false	false
"neo4j"	[]	"read-write"	"localhost:7687"	"standalone"	"online"	"online"	""	true	true
"system"	[]	"read-write"	"localhost:7687"	"standalone"	"online"	"online"	""	false	false

Rows: 4

### 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.

### Example 440. CREATE DATABASE

Query

```
CREATE DATABASE customers IF NOT EXISTS
```

## Example 441. CREATE OR REPLACE DATABASE

### Query

```
CREATE OR REPLACE DATABASE customers
```

This is equivalent to running `DROP DATABASE customers IF EXISTS` followed by `CREATE DATABASE customers`.



The `IF NOT EXISTS` and `OR REPLACE` parts of this command cannot be used together.

### Options Enterprise edition

The create database command can have a map of options, e.g. `OPTIONS {key: 'value'}`.

Key	Value	Description
<code>existingData</code>	<code>use</code>	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.
<code>existingDataSeedInstance</code>	instance ID of the cluster node	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.



The `existingData` and `existingDataSeedInstance` options cannot be combined with the `OR REPLACE` part of this command.

### Altering databases Enterprise edition

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.

#### Example 442. ALTER DATABASE

##### Query

```
ALTER DATABASE customers SET ACCESS READ ONLY
```

##### Result

```
System updates: 1  
Rows: 0
```

#### Example 443. SHOW DATABASES

The database access mode can be seen in the `access` output column of the command `SHOW DATABASES`.

##### Query

```
SHOW DATABASES yield name, access
```

##### Table 512. Result

name	access
"customers"	"read-only"
"movies"	"read-write"
"neo4j"	"read-write"
"system"	"read-write"

Rows: 4

#### Example 444. ALTER DATABASE

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.

##### Query

```
ALTER DATABASE nonExisting IF EXISTS  
SET ACCESS READ WRITE
```

## Stopping databases Enterprise edition

Databases can be stopped using the command `STOP DATABASE`.

### Example 445. STOP DATABASE

#### Query

```
STOP DATABASE customers
```

#### Result

```
System updates: 1  
Rows: 0
```

### Example 446. SHOW DATABASE

The status of the stopped database can be seen using the command `SHOW DATABASE name`.

#### Query

```
SHOW DATABASE customers
```

Table 513. Result

name	aliases	access	address	role	requested Status	currentStatus	error	default	home
"customers"	[]	"read-only"	"localhost:7687"	"standalone"	"offline"	"offline"	""	false	false

Rows: 1

## Starting databases Enterprise edition

Databases can be started using the command `START DATABASE`.

### Example 447. START DATABASE

#### Query

```
START DATABASE customers
```

#### Result

```
System updates: 1  
Rows: 0
```

### Example 448. SHOW DATABASE

The status of the started database can be seen using the command `SHOW DATABASE name`.

#### Query

```
SHOW DATABASE customers
```

#### Table 514. Result

name	aliases	access	address	role	requested Status	currentStatus	error	default	home
"customers"	[]	"read-only"	"localhost:7687"	"standalone"	"online"	"online"	" "	false	false

Rows: 1

## Deleting databases [Enterprise edition](#)

Databases can be deleted using the command `DROP DATABASE`.

### Example 449. DROP DATABASE

#### Query

```
DROP DATABASE customers
```

#### Result

```
System updates: 1  
Rows: 0
```

### Example 450. SHOW DATABASES

When a database has been deleted, it will no longer show up in the listing provided by the command `SHOW DATABASES`.

#### Query

```
SHOW DATABASES
```

Table 515. Result

name	aliases	access	address	role	requested Status	currentStatus	error	default	home
"movies"	["films", "motion pictures"]	"read-write"	"localhost:7687"	"standalone"	"online"	"online"	" "	false	false
"neo4j"	[]	"read-write"	"localhost:7687"	"standalone"	"online"	"online"	" "	true	true
"system"	[]	"read-write"	"localhost:7687"	"standalone"	"online"	"online"	" "	false	false

Rows: 3

### Example 451. DROP DATABASE

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

```
DROP DATABASE customers IF EXISTS
```

The `DROP DATABASE` command will remove a database entirely.

## Example 452. DROP DATABASE

You can request that a dump of the store files is produced first, and stored in the path configured using the `dbms.directories.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

```
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

```
DROP DATABASE customers IF EXISTS DUMP DATA
```

## Wait options Enterprise edition

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.



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.



## Example 453. CREATE DATABASE

### Query

```
CREATE DATABASE slow WAIT 5 SECONDS
```

### Table 516. Result

address	state	message	success
"localhost:7687"	"CaughtUp"	"caught up"	true

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.

## Database alias management

*This section 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 requires 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.

Database 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:



More details about the syntax descriptions can be found [here](#).

Table 517. Alias management command syntax

Command	Syntax
SHOW ALIASES	<p>Lists both local and remote database aliases.</p> <pre>SHOW ALIASES FOR DATABASE[S] [WHERE expression]</pre> <pre>SHOW ALIASES FOR DATABASE[S] YIELD { *   field[, ...] } [ORDER BY field[, ...]] [SKIP n] [LIMIT n] [WHERE expression] [RETURN field[, ...]] [ORDER BY field[, ...]] [SKIP n] [LIMIT n]</pre>
CREATE ALIAS	<p>Create a local database alias.</p> <pre>CREATE ALIAS name [IF NOT EXISTS] FOR DATABASE targetName</pre> <pre>CREATE OR REPLACE ALIAS name FOR DATABASE targetName</pre>
CREATE ALIAS ... AT	<p>Create a remote database alias.</p> <pre>CREATE ALIAS name [IF NOT EXISTS] FOR DATABASE targetName AT 'url' USER username PASSSWORd 'password' [DRIVER "{" setting: value[, ...] "}"]</pre> <pre>CREATE OR REPLACE ALIAS name FOR DATABASE targetName AT 'url' USER username PASSSWORd 'password' [DRIVER "{" setting: value[, ...] "}"]</pre>
ALTER ALIAS	<p>Alter a local database alias.</p> <pre>ALTER ALIAS name [IF EXISTS] SET DATABASE TARGET targetName</pre>
ALTER ALIAS ...	<p>Alter a remote database alias.</p> <pre>ALTER ALIAS name [IF EXISTS] SET DATABASE [TARGET targetName AT 'url'] [USER username] [PASSSWORd 'password'] [DRIVER "{" setting: value[, ...] "}"]</pre>
DROP ALIAS	<p>Drop either a local or remote database alias.</p> <pre>DROP ALIAS name [IF EXISTS] FOR DATABASE</pre>

This is the list of the allowed driver settings for remote aliases.

Table 518. `ssl_enforced`

Description	SSL for remote alias drivers is configured through the target url scheme. If <code>ssl_enforced</code> is set to true, a secure url scheme is enforced. This will be validated when the command is executed.
Valid values	Boolean
Default value	true

Table 519. `connection_timeout`

Description	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.
Valid values	Duration
Default value	<a href="#">dbms.routing.driver.connection.connect_timeout</a>

Table 520. `connection_max_lifetime`

Description	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).
Valid values	Duration.  Zero and negative values result in lifetime not being checked.
Default value	<a href="#">dbms.routing.driver.connection.max_lifetime</a>

Table 521. `connection_pool_acquisition_timeout`

Description	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.
-------------	--

Valid values	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.
Default value	<code>dbms.routing.driver.connection.pool.acquisition_timeout</code>

Table 522. `connection_pool_max_size`

Description	Maximum total number of connections to be managed by a connection pool. The limit is enforced for a combination of a host and user.
Valid values	Integer.  Negative values are allowed and result in unlimited pool. Value of 0 is not allowed.
Default value	<code>dbms.routing.driver.connection.pool.max_size</code>

Table 523. `logging_level`

Description	Sets level for driver internal logging.
Valid values	<code>org.neo4j.logging.Level</code> .  One of <code>DEBUG</code> , <code>INFO</code> , <code>WARN</code> , <code>ERROR</code> , or <code>NONE</code> .
Default value	<code>dbms.routing.driver.logging.level</code>



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.

## Listing database aliases Enterprise edition

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:

Column	Description
<code>name</code>	The name of the database alias. <span style="border: 1px solid #0070C0; border-radius: 5px; padding: 2px 5px; color: #0070C0;">Default output</span>

Column	Description
<code>database</code>	The names of the target database. <span>Default output</span>
<code>location</code>	The location of the database, either <code>local</code> or <code>remote</code> . <span>Default output</span>
<code>url</code>	Target location or <code>null</code> if the target is local. <span>Default output</span>
<code>user</code>	User connecting to the remote database or <code>null</code> if the target database is local. <span>Default output</span>
<code>driver</code>	The driver options for connection to the remote database or <code>null</code> if the target database is local or if no driver settings are added. List of <a href="#">driver settings</a> allowed for remote database aliases.
<code>MAP</code>	<code>properties</code>
Any properties set on the database alias.	<code>MAP</code>

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.

#### Example 454. SHOW ALIASES FOR DATABASE

A summary of all available databases alias can be displayed using the command `SHOW ALIASES FOR DATABASE`.

#### Query

```
SHOW ALIASES FOR DATABASE
```

Table 524. Result

name	database	location	url	user
<code>"films"</code>	<code>"movies"</code>	<code>"local"</code>	<code>&lt;null&gt;</code>	<code>&lt;null&gt;</code>
<code>"motion pictures"</code>	<code>"movies"</code>	<code>"local"</code>	<code>&lt;null&gt;</code>	<code>&lt;null&gt;</code>
<code>"movie scripts"</code>	<code>"scripts"</code>	<code>"remote"</code>	<code>"neo4j+s://location:7687"</code>	<code>"alice"</code>

Rows: 3

### Example 455. SHOW ALIASES FOR DATABASE

#### Query

```
SHOW ALIASES FOR DATABASE YIELD *
```

#### Table 525. Result

name	database	location	url	user	driver
"films"	"movies"	"local"	<null>	<null>	<null>
"motion pictures"	"movies"	"local"	<null>	<null>	<null>
"movie scripts"	"scripts"	"remote"	"neo4j+s://location:7687"	"alice"	{connection_pool_max_size -> 10, connection_pool_idle_test -> PT2M, connection_pool_acquisition_timeout -> PT1M, connection_max_lifetime -> PT1H, logging_level -> "INFO", ssl_enforced -> true, connection_timeout -> PT5S}

Rows: 3

### Example 456. SHOW ALIASES FOR DATABASE

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 526. Result

count
3

Rows: 1

## Example 457. SHOW ALIASES FOR DATABASE

It is 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'.
- 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 527. Result

name	url	database
"motion pictures"	<null>	"movies"
"movie scripts"	"neo4j+s://location:7687"	"scripts"

Rows: 2

## Creating database aliases Enterprise edition

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.

## Creating local database aliases Enterprise edition

Local aliases are created with a target database.

### Example 458. CREATE ALIAS

Query

```
CREATE ALIAS `northwind` FOR DATABASE `northwind-graph-2021`
```

System updates: 1  
Rows: 0

### Example 459. SHOW DATABASE

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 528. Result

name	aliases	access	address	role	requested Status	currentStatus	error	default	home
"northwind-graph-2021"	["northwind"]	"read-write"	"localhost:7687"	"standalone"	"online"	"online"	"	false	false

Rows: 1



### Example 460. SHOW ALIASES FOR DATABASE

#### Query

```
SHOW ALIASES FOR DATABASE
WHERE name = 'northwind'
```

#### Table 529. Result

name	database	location	url	user
"northwind"	"northwind-graph-2021"	"local"	<null>	<null>

Rows: 1

### Example 461. CREATE ALIAS

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`
```

Rows: 0

### Example 462. CREATE OR REPLACE ALIAS

It is 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`
```

System updates: 2  
Rows: 0

This is equivalent to running:

#### Query

```
DROP ALIAS `northwind` IF EXISTS FOR DATABASE
CREATE ALIAS `northwind` FOR DATABASE `northwind-graph-2020`
```

## Creating remote database aliases Enterprise edition

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.

Creating remote aliases also allows `IF NOT EXISTS` and `OR REPLACE` clauses. Both check for any remote or local database aliases.

#### Example 463. CREATE ALIAS

##### Query

```
CREATE ALIAS `remote-northwind` FOR DATABASE `northwind-graph-2020`
AT "neo4j+s://location:7687"
USER alice
PASSWORD 'example_secret'
```

```
System updates: 1
Rows: 0
```

#### Example 464. CREATE ALIAS

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](#).

##### Query

```
CREATE ALIAS `remote-with-driver-settings` FOR DATABASE `northwind-graph-2020`
AT "neo4j+s://location:7687"
USER alice
PASSWORD 'example_secret'
DRIVER {
  connection_timeout: duration({minutes: 1}),
  connection_pool_max_size: 10
}
```

```
System updates: 1
Rows: 0
```

#### Example 465. SHOW ALIASES FOR DATABASE

When a database alias pointing to a remote database has been created, its details can be shown with the `SHOW ALIASES FOR DATABASE` command.

##### Query

```
SHOW ALIASES FOR DATABASE
WHERE name = 'remote-northwind'
```

##### Table 530. Result

name	database	location	url	user
"remote-northwind"	"northwind-graph-2020"	"remote"	"neo4j+s://location:7687"	"alice"

Rows: 1

## Example 466. SHOW ALIASES FOR DATABASE

### Query

```
SHOW ALIASES FOR DATABASE YIELD *  
WHERE name = 'remote-with-driver-settings'
```

### Table 531. Result

name	database	location	url	user	driver
"remote-with-driver-settings"	"northwind-graph-2020"	"remote"	"neo4j+s://location:7687"	"alice"	{connection_pool_max_size -> 10, connection_timeout -> PT1M}

Rows: 1

## Altering database aliases Enterprise edition

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 467. ALTER ALIAS

Example of altering a local database alias target.

### Query

```
ALTER ALIAS `northwind`  
SET DATABASE TARGET `northwind-graph-2021`
```

System updates: 1  
Rows: 0

## Example 468. ALTER ALIAS

Example of altering a remote database alias target.

### Query

```
ALTER ALIAS `remote-northwind` SET DATABASE  
TARGET `northwind-graph-2020` AT "neo4j+s://other-location:7687"
```

System updates: 1  
Rows: 0

## Example 469. ALTER ALIAS

Example of altering a remote alias credentials and driver settings.

### Query

```
ALTER ALIAS `remote-with-driver-settings` SET DATABASE
USER bob
PASSWORD 'new_example_secret'
DRIVER {
  connection_timeout: duration({ minutes: 1}),
  logging_level: 'debug'
}
```

System updates: 1  
Rows: 0



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 470. ALTER ALIAS

Example of altering a remote alias to remove all custom driver settings.

### Query

```
ALTER ALIAS `movie scripts` SET DATABASE
DRIVER {}
```

System updates: 1  
Rows: 0

### Example 471. SHOW DATABASE

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

```
SHOW DATABASE `northwind`
```

Table 532. Result

name	aliases	access	address	role	requested Status	currentStatus	error	default	home
"northwind-graph-2021"	["northwind"]	"read-write"	"localhost:7687"	"standalone"	"online"	"online"	""	false	false

Rows: 1

### Example 472. SHOW ALIASES FOR DATABASE

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 533. Result

name	database	location	url	user	driver
"movie scripts"	"scripts"	"remote"	"neo4j+s://location:7687"	"alice"	{}
"northwind"	"northwind-graph-2021"	"local"	<null>	<null>	<null>
"remote-northwind"	"northwind-graph-2020"	"remote"	"neo4j+s://other-location:7687"	"alice"	{}
"remote-with-driver-settings"	"northwind-graph-2020"	"remote"	"neo4j+s://location:7687"	"bob"	{logging_level -> "DEBUG", connection_timeout -> PT1M}

Rows: 4

### Example 473. ALTER ALIAS

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`
```

Rows: 0

## Deleting database aliases Enterprise edition

Both local and remote aliases can be deleted using the **DROP ALIAS** command. The required privileges are described [here](#).

### Example 474. DROP ALIAS

Drop a local database alias.

#### Query

```
DROP ALIAS `northwind` FOR DATABASE
```

System updates: 1  
Rows: 0

### Example 475. DROP ALIAS

Drop a remote database alias.

#### Query

```
DROP ALIAS `remote-northwind` FOR DATABASE
```

System updates: 1  
Rows: 0

### Example 476. SHOW DATABASE

When a database alias has been deleted, it will no longer show up in the aliases column provided by the command `SHOW DATABASES`.

Query

```
SHOW DATABASE `northwind-graph-2021`
```

Table 534. Result

name	aliases	access	address	role	requested Status	currentStatus	error	default	home
"northwind-graph-2021"	[]	"read-write"	"localhost:7687"	"standalone"	"online"	"online"	""	false	false

Rows: 1

### Example 477. SHOW ALIASES FOR DATABASE

When a database alias has been deleted, it will no longer show up in the aliases column provided by the command `SHOW ALIASES FOR DATABASE`.

List all database aliases.

Query

```
SHOW ALIASES FOR DATABASE
```

Table 535. Result

name	database	location	url	user
"films"	"movies"	"local"	<null>	<null>
"motion pictures"	"movies"	"local"	<null>	<null>
"movie scripts"	"scripts"	"remote"	"neo4j+s://location:7687"	"alice"
"remote-with-driver-settings"	"northwind-graph-2020"	"remote"	"neo4j+s://location:7687"	"bob"

Rows: 4

## Example 478. DROP ALIAS

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.

Query

```
DROP ALIAS `northwind` IF EXISTS FOR DATABASE
```

Rows: 0

## Access control

### Overview

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](#).

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 or removing these access rights.

In addition, any newly created roles can be assigned to any combination of *privileges*, so that you may set specific access controls for them. 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.

### Categories of access control

More details about specific categories of access control can be found in the following sections:

- [Managing users](#)
- [Managing roles](#)
- [Managing privileges](#)
- [Built-in roles and privileges](#)
- [Read privileges](#)



- [Write privileges](#)
- [Database administration](#)
- [DBMS administration](#)
- [Limitations](#)

## 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.

### User states

There are two types of user states in the `system` database:

- **ACTIVE** state (default for new users): Users can log into Neo4j and perform queries according to their privileges.
- **SUSPENDED** state Enterprise edition:
  - Native users who authenticate and authorize against the system graph cannot log into Neo4j. If suspended while using Neo4j, they lose all assigned roles with their privileges, including the **PUBLIC** role, until reactivated.
  - Users who authenticate and authorize against an external ID provider (e.g., LDAP) can still log in. If suspended while using Neo4j, they retain the roles and the privileges assigned by the external provider, including the **PUBLIC** role. To prevent any of these, you need to use the mechanisms of their identity provider.

### User management command syntax



More details about the syntax descriptions can be found [here](#).

Command	<code>SHOW CURRENT USER</code>
Syntax	<pre>SHOW CURRENT USER [YIELD { *   field[, ...] } [ORDER BY field[, ...]] [SKIP n] [LIMIT n]] [WHERE expression] [RETURN field[, ...] [ORDER BY field[, ...]] [SKIP n] [LIMIT n]]</pre>
Description	<p>Lists the current user.</p> <p>When using the <code>RETURN</code> clause, the <code>YIELD</code> clause is mandatory and must not be omitted.</p> <p>For more information, see <a href="#">Listing current user</a>.</p>

Required privilege	None
Command	<b>SHOW USERS</b>
Syntax	<pre>SHOW USERS [YIELD { *   field[, ...] } [ORDER BY field[, ...]] [SKIP n] [LIMIT n]] [WHERE expression] [RETURN field[, ...] [ORDER BY field[, ...]] [SKIP n] [LIMIT n]]</pre>
Description	<p>Lists all users.</p> <p>When using the <b>RETURN</b> clause, the <b>YIELD</b> clause is mandatory and must not be omitted.</p> <p>For more information, see <a href="#">Listing users</a>.</p>
Required privilege	<pre>GRANT SHOW USER</pre> <p>(see <a href="#">DBMS USER MANAGEMENT privileges</a>)</p>
Command	<b>SHOW USER PRIVILEGES</b>
Syntax	<pre>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]]</pre>
Description	<p>Lists the privileges granted to the specified users or the current user if no user is specified.</p> <p>When using the <b>RETURN</b> clause, the <b>YIELD</b> clause is mandatory and must not be omitted.</p> <p>For more information, see <a href="#">Listing privileges</a>.</p>
Required privilege	<pre>GRANT SHOW PRIVILEGE</pre> <p>(see <a href="#">DBMS PRIVILEGE MANAGEMENT privileges</a>)</p> <pre>GRANT SHOW USER</pre> <p>(see <a href="#">DBMS USER MANAGEMENT privileges</a>)</p>
Command	<b>CREATE USER</b>

Syntax	<pre>CREATE USER name [IF NOT EXISTS] SET [PLAINTEXT   ENCRYPTED] PASSWORD 'password' [[SET PASSWORD] CHANGE [NOT] REQUIRED] [SET STATUS {ACTIVE   SUSPENDED}] [SET HOME DATABASE name]</pre>
Description	<p>Creates a new user.</p> <p>For more information, see <a href="#">Creating users</a>.</p>
Required privilege	<pre>GRANT CREATE USER</pre> <p>(see <a href="#">DBMS USER MANAGEMENT privileges</a>)</p>

Command	<p><b>CREATE OR REPLACE USER</b></p>
Syntax	<pre>CREATE OR REPLACE USER name SET [PLAINTEXT   ENCRYPTED] PASSWORD 'password' [[SET PASSWORD] CHANGE [NOT] REQUIRED] [SET STATUS {ACTIVE   SUSPENDED}] [SET HOME DATABASE name]</pre>
Description	<p>Creates a new user, or if a user with the same name exists, replace it.</p> <p>For more information, see <a href="#">Creating users</a>.</p>
Required privilege	<pre>GRANT CREATE USER</pre> <p>(see <a href="#">DBMS USER MANAGEMENT privileges</a>)</p> <pre>GRANT DROP USER</pre> <p>(see <a href="#">DBMS USER MANAGEMENT privileges</a>)</p>

Command	<p><b>RENAME USER</b></p>
Syntax	<pre>RENAME USER name [IF EXISTS] TO otherName</pre>
Description	<p>Changes the name of a user.</p> <p>For more information, see <a href="#">Renaming users</a>.</p>

Required privilege	<div style="border: 1px solid #ccc; padding: 5px; margin-bottom: 10px;">GRANT RENAME USER</div> <p>(see <a href="#">DBMS USER MANAGEMENT privileges</a>)</p>
--------------------	--

Command	<b>ALTER USER</b>
Syntax	<pre>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]</pre>
Description	<p>Modifies the settings for an existing user. At least one <b>SET</b> or <b>REMOVE</b> clause is required. <b>SET</b> and <b>REMOVE</b> clauses cannot be combined in the same command.</p> <p>For more information, see <a href="#">Modifying users</a>.</p>

Required privilege	<div style="border: 1px solid #ccc; padding: 5px; margin-bottom: 10px;">GRANT SET PASSWORD</div> <p>[source, privilege, role="noheader"</p> <div style="border: 1px solid #ccc; padding: 5px; margin-bottom: 10px;">GRANT SET USER STATUS</div> <div style="border: 1px solid #ccc; padding: 5px; margin-bottom: 10px;">GRANT SET USER HOME DATABASE</div> <p>(see <a href="#">DBMS USER MANAGEMENT privileges</a>)</p>
--------------------	---

Command	<b>ALTER CURRENT USER SET PASSWORD</b>
Syntax	<div style="border: 1px solid #ccc; padding: 5px;">ALTER CURRENT USER SET PASSWORD FROM 'oldPassword' TO 'newPassword'</div>
Description	<p>Changes the current user's password.</p> <p>For more information, see <a href="#">Changing the current user's password</a>.</p>
Required privilege	None

Command	<b>DROP USER</b>
Syntax	<div style="border: 1px solid #ccc; padding: 5px;">DROP USER name [IF EXISTS]</div>

Description	Removes an existing user.  For more information, see <a href="#">Delete users</a> .
Required privilege	GRANT DROP USER  (see <a href="#">DBMS USER MANAGEMENT privileges</a> )



The `SHOW USER[S] PRIVILEGES` command is only available in Neo4j Enterprise Edition.  
[Enterprise edition](#)

## Listing current user

The currently logged-in user can be seen using `SHOW CURRENT USER`, which will produce a table with the following columns:

Column	Description	Community Edition	Enterprise Edition
user	User name	✓	✓
roles	Roles granted to the user.	✗	✓
passwordChangeRequired	If <code>true</code> , the user must change their password at the next login.	✓	✓
suspended	If <code>true</code> , the user is currently suspended (cannot log in).	✗	✓
home	The home database configured by the user, or <code>null</code> 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.	✗	✓

```
SHOW CURRENT USER
```

Table 536. Result

user	roles	passwordChangeRequired	suspended	home
"jake"	["PUBLIC"]	false	false	<null>
Rows: 1				



This command is only supported for a logged-in user and will return an empty result if authorization has been disabled.

## Listing users

Available users can be seen using `SHOW USERS`, which will produce a table of users with the following columns:

Column	Description	Community Edition	Enterprise Edition
user	User name	✓	✓
roles	Roles granted to the user.	✗	✓
passwordChangeRequired	If <code>true</code> , the user must change their password at the next login.	✓	✓
suspended	If <code>true</code> , the user is currently suspended (cannot log in).	✗	✓
home	The home database configured by the user, or <code>null</code> if no home database has been configured. A home database will be resolved if it is either pointing to a database or a database alias. If this database is unavailable and the user does not specify a database to use, they will not be able to log in.	✗	✓

```
SHOW USERS
```

Table 537. Result

user	roles	passwordChangeRequired	suspended	home
"neo4j"	["admin", "PUBLIC"]	true	false	<null>

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 dbms set-initial-password <password>`, otherwise it is necessary to change the password after the first login.

### Example 479. 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 480. Show user

It is possible to add a `RETURN` clause to further manipulate the results after filtering. In this example, the `RETURN` clause is used to filter out the `roles` column and rename the `user` column to `adminUser`.

```
SHOW USERS YIELD roles, user
WHERE 'admin' IN roles
RETURN user AS adminUser
```



The `SHOW USER name PRIVILEGES` command is described in [Listing privileges](#).

## Creating users

Users can be created using `CREATE USER`.

```
CREATE USER name [IF NOT EXISTS]
SET [PLAINTEXT | ENCRYPTED] PASSWORD 'password'
[[SET PASSWORD] CHANGE [NOT] REQUIRED]
[SET STATUS {ACTIVE | SUSPENDED}]
[SET HOME DATABASE name]
```

Users can be created or replaced using `CREATE OR REPLACE USER`.

```
CREATE OR REPLACE USER name
SET [PLAINTEXT | ENCRYPTED] PASSWORD 'password'
[[SET PASSWORD] CHANGE [NOT] REQUIRED]
[SET STATUS {ACTIVE | SUSPENDED}]
[SET HOME DATABASE name]
```

- 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 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 → 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.

- 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.



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 481. 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:

```
CREATE USER jake
SET PASSWORD 'abc' CHANGE REQUIRED
SET STATUS SUSPENDED
SET HOME DATABASE anotherDb
```

#### Example 482. 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:

```
CREATE USER jake
SET ENCRYPTED PASSWORD
'1,6d57a5e0b3317055454e455f96c98c750c77fb371f3f0634a1b8ff2a55c5b825,190ae47c661e0668a0c8be8a21ff78a4a
34cdf918cae3c407e907b73932bd16c' CHANGE NOT REQUIRED
SET STATUS ACTIVE
```



The `SET STATUS {ACTIVE | SUSPENDED}` and `SET HOME DATABASE` parts of the commands are only available in Neo4j Enterprise Edition. [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 483. 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 484. 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`.

## Renaming users

Users can be renamed with the `RENAME USER` command.

```
RENAME USER jake TO bob
```

```
SHOW USERS
```

Table 538. Result

user	roles	passwordChangeRequired	suspended	home
"bob"	["PUBLIC"]	true	false	<null>
"neo4j"	["admin", "PUBLIC"]	true	false	<null>

Rows: 2



The `RENAME USER` command is only available when using native authentication and authorization.

## Modifying users

Users can be modified with `ALTER USER`.

```
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. [Enterprise edition](#)

The changes to the user will appear on the list provided by `SHOW USERS`:

```
SHOW USERS
```

Table 539. Result

user	roles	passwordChangeRequired	suspended	home
"bob"	["PUBLIC"]	false	false	<null>
"neo4j"	["admin", "PUBLIC"]	true	false	<null>

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'
```

## 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.

## Delete users

Users can be deleted with `DROP USER`.

```
DROP USER bob
```

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 540. Result

user	roles	passwordChangeRequired	suspended	home
"neo4j"	["admin", "PUBLIC"]	true	false	<null>
Rows: 1				

## Managing roles

*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.

### Role management command syntax



More details about the syntax descriptions can be found [here](#).

Command	<code>SHOW ROLES</code>
Syntax	<pre>SHOW [ALL POPULATED] ROLES [YIELD { *   field[, ...] } [ORDER BY field[, ...]] [SKIP n] [LIMIT n]] [WHERE expression] [RETURN field[, ...] [ORDER BY field[, ...]] [SKIP n] [LIMIT n]]</pre>
Description	<p>Lists roles.</p> <p>When using the <code>RETURN</code> clause, the <code>YIELD</code> clause is mandatory and must not be omitted.</p> <p>For more information, see <a href="#">Listing roles</a>.</p>

Required privilege	<div style="border: 1px solid #ccc; padding: 5px; margin-bottom: 10px;">GRANT SHOW ROLE</div> <p>(see <a href="#">DBMS ROLE MANAGEMENT privileges</a>).</p>
--------------------	---

Command	<b>SHOW ROLES WITH USERS</b>
Syntax	<pre>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]]</pre>
Description	<p>Lists roles and users assigned to them.</p> <p>When using the <b>RETURN</b> clause, the <b>YIELD</b> clause is mandatory and must not be omitted.</p> <p>For more information, see <a href="#">Listing roles</a>.</p>
Required privilege	<div style="border: 1px solid #ccc; padding: 5px; margin-bottom: 10px;">GRANT SHOW ROLE</div> <p>(see <a href="#">DBMS ROLE MANAGEMENT privileges</a>)</p> <div style="border: 1px solid #ccc; padding: 5px; margin-bottom: 10px;">GRANT SHOW USER</div> <p>(see <a href="#">DBMS USER MANAGEMENT privileges</a>)</p>

Command	<b>SHOW ROLE PRIVILEGES</b>
Syntax	<pre>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]]</pre>
Description	<p>Lists the privileges granted to the specified roles.</p> <p>When using the <b>RETURN</b> clause, the <b>YIELD</b> clause is mandatory and must not be omitted.</p> <p>For more information, see <a href="#">Listing privileges</a>.</p>
Required privilege	<div style="border: 1px solid #ccc; padding: 5px; margin-bottom: 10px;">GRANT SHOW PRIVILEGE</div> <p>(see <a href="#">DBMS PRIVILEGE MANAGEMENT privileges</a>)</p>

Command	<b>CREATE ROLE</b>
---------	--------------------

<b>Syntax</b>	<code>CREATE ROLE name [IF NOT EXISTS] [AS COPY OF otherName]</code>
<b>Description</b>	Creates a new role.  For more information, see <a href="#">Creating roles</a> .
<b>Required privilege</b>	<code>GRANT CREATE ROLE</code>  (see <a href="#">DBMS ROLE MANAGEMENT privileges</a> )

<b>Command</b>	<b>CREATE OR REPLACE ROLE</b>
<b>Syntax</b>	<code>CREATE OR REPLACE ROLE name [AS COPY OF otherName]</code>
<b>Description</b>	Creates a new role, or if a role with the same name exists, replace it.  For more information, see <a href="#">Creating roles</a> .
<b>Required privilege</b>	<code>GRANT CREATE ROLE</code>  <code>GRANT DROP ROLE</code>  (see <a href="#">DBMS ROLE MANAGEMENT privileges</a> )

<b>Command</b>	<b>RENAME ROLE</b>
<b>Syntax</b>	<code>RENAME ROLE name [IF EXISTS] TO otherName</code>
<b>Description</b>	Changes the name of a role.  For more information, see <a href="#">Renaming roles</a> .
<b>Required privilege</b>	<code>GRANT RENAME ROLE</code>  (see <a href="#">DBMS ROLE MANAGEMENT privileges</a> )

<b>Command</b>	<b>DROP ROLE</b>
----------------	------------------

Syntax	<code>DROP ROLE name [IF EXISTS]</code>
Description	Removes a role.  For more information, see <a href="#">Deleting roles</a> .

Command	<code>GRANT ROLE TO</code>
Syntax	<code>GRANT ROLE[S] name[, ...] TO user[, ...]</code>
Description	Assigns roles to users.  For more information, see <a href="#">Assigning roles to users</a> .
Required privilege	<code>GRANT ASSIGN ROLE</code>  (see <a href="#">DBMS ROLE MANAGEMENT privileges</a> )

Command	<code>REVOKE ROLE</code>
Syntax	<code>REVOKE ROLE[S] name[, ...] FROM user[, ...]</code>
Description	Removes roles from users.  For more information, see <a href="#">Revoking roles from users</a> .
Required privilege	<code>GRANT REMOVE ROLE</code>  (see <a href="#">DBMS ROLE MANAGEMENT privileges</a> )

## 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 541. Result

role
"PUBLIC"
"admin"
"architect"
"editor"
"publisher"
"reader"
Rows: 6

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 542. Result

role	member
"PUBLIC"	"neo4j"
"PUBLIC"	"bob"
"PUBLIC"	"user1"
"PUBLIC"	"user2"
"PUBLIC"	"user3"
"admin"	"neo4j"
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 543. Result

role
"editor"
"publisher"
"reader"

Rows: 3



The **SHOW ROLE name PRIVILEGES** command is found in [Listing privileges](#).

## 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 485. Copy a role

```
CREATE ROLE mysecondrole AS COPY OF myrole
```

Created roles will appear on the list provided by `SHOW ROLES`.

Example 486. List roles

```
SHOW ROLES
```

Table 544. Result

role
"PUBLIC"
"admin"
"architect"
"editor"
"myrole"
"mysecondrole"
"publisher"
"reader"

Rows: 8

The `CREATE ROLE` command is optionally idempotent, with the default behavior to throw an exception if the role already exists. Adding `IF NOT EXISTS` to the `CREATE ROLE` command will ensure that no exception is thrown and nothing happens should the role already exist.

Example 487. Create role if not exists

```
CREATE ROLE myrole IF NOT EXISTS
```

The `CREATE OR REPLACE ROLE` command will result in any existing role being deleted and a new one created.

Example 488. Create or replace role

```
CREATE OR REPLACE ROLE myrole
```

This is equivalent to running `DROP ROLE myrole IF EXISTS` followed by `CREATE ROLE myrole`.



- The `CREATE OR REPLACE ROLE` command does not allow you to use the `IF NOT EXISTS`.

## Renaming roles

Roles can be renamed using `RENAME ROLE` command:

```
RENAME ROLE mysecondrole TO mythirdrole
```

```
SHOW ROLES
```

Table 545. Result

role
"PUBLIC"
"admin"
"architect"
"editor"
"myrole"
"mythirdrole"
"publisher"
"reader"

Rows: 8



The `RENAME ROLE` command is only available when using native authentication and authorization.

## 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 USERS
```

Table 546. Result

user	roles	passwordChangeRequired	suspended	home
"bob"	["myrole", "PUBLIC"]	false	false	<null>
"neo4j"	["admin", "PUBLIC"]	true	false	<null>
"user1"	["PUBLIC"]	true	false	<null>
"user2"	["PUBLIC"]	true	false	<null>
"user3"	["PUBLIC"]	true	false	<null>

user	roles	passwordChangeRequired	suspended	home
Rows: 5				

It is possible to assign multiple roles to multiple users in one command:

```
GRANT ROLES role1, role2 TO user1, user2, user3
```

```
SHOW USERS
```

Table 547. Result

user	roles	passwordChangeRequired	suspended	home
"bob"	["myrole", "PUBLIC"]	false	false	<null>
"neo4j"	["admin", "PUBLIC"]	true	false	<null>
"user1"	["role1", "role2", "PUBLIC"]	true	false	<null>
"user2"	["role1", "role2", "PUBLIC"]	true	false	<null>
"user3"	["role1", "role2", "PUBLIC"]	true	false	<null>
Rows: 5				

## 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 USERS
```

Table 548. Result

user	roles	passwordChangeRequired	suspended	home
"bob"	["PUBLIC"]	false	false	<null>
"neo4j"	["admin", "PUBLIC"]	true	false	<null>
"user1"	["role1", "role2", "PUBLIC"]	true	false	<null>
"user2"	["role1", "role2", "PUBLIC"]	true	false	<null>
"user3"	["role1", "role2", "PUBLIC"]	true	false	<null>
Rows: 5				

It is possible to revoke multiple roles from multiple users in one command:

```
REVOKE ROLES role1, role2 FROM user1, user2, user3
```

## Deleting roles

Roles can be deleted using `DROP ROLE` command:

```
DROP ROLE mythirdrole
```

When a role has been deleted, it will no longer appear on the list provided by `SHOW ROLES`:

```
SHOW ROLES
```

Table 549. Result

role
"PUBLIC"
"admin"
"architect"
"editor"
"myrole"
"publisher"
"reader"
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
```

## 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](#).



More details about the syntax descriptions can be found [here](#).

## Graph privilege commands (**GRANT**, **DENY** and **REVOKE**) Enterprise edition

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.



If you delete a database and create a new one with the same name, the new one will **NOT** have the privileges previously assigned to the deleted 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 550. General grant ON GRAPH privilege syntax

Command	<b>GRANT ... ON ... TO ...</b>
Syntax	<code>GRANT graph-privilege ON { HOME GRAPH   GRAPH[S] { *   name[, ...] } } [entity] TO role[, ...]</code>
Description	Grants a privilege to one or multiple roles.

Table 551. General deny ON GRAPH privilege syntax

Command	<b>DENY ... ON ... TO ...</b>
Syntax	<code>DENY graph-privilege ON { HOME GRAPH   GRAPH[S] { *   name[, ...] } } [entity] TO role[, ...]</code>
Description	Denies a privilege to one or multiple roles.

Table 552. General revoke ON GRAPH privilege syntax

Command	<b>REVOKE GRANT ... ON ... FROM ...</b>
Syntax	<code>REVOKE GRANT graph-privilege ON { HOME GRAPH   GRAPH[S] { *   name[, ...] } } [entity] FROM role[, ...]</code>
Description	Revokes a granted privilege from one or multiple roles.

Table 553. General revoke ON GRAPH privilege syntax

Command	<b>REVOKE DENY ... ON ... FROM ...</b>
---------	--

Syntax	<pre>REVOKE DENY graph-privilege ON { HOME GRAPH   GRAPH[S] { *   name[, ...] } } [entity] FROM role[, ...]</pre>
Description	Revokes a denied privilege from one or multiple roles.

Table 554. General revoke ON GRAPH privilege syntax

Command	REVOKE ... ON ... FROM ...
Syntax	<pre>REVOKE graph-privilege ON { HOME GRAPH   GRAPH[S] { *   name[, ...] } } [entity] FROM role[, ...]</pre>
Description	Revokes a granted or denied privilege from one or multiple roles.

i

**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:

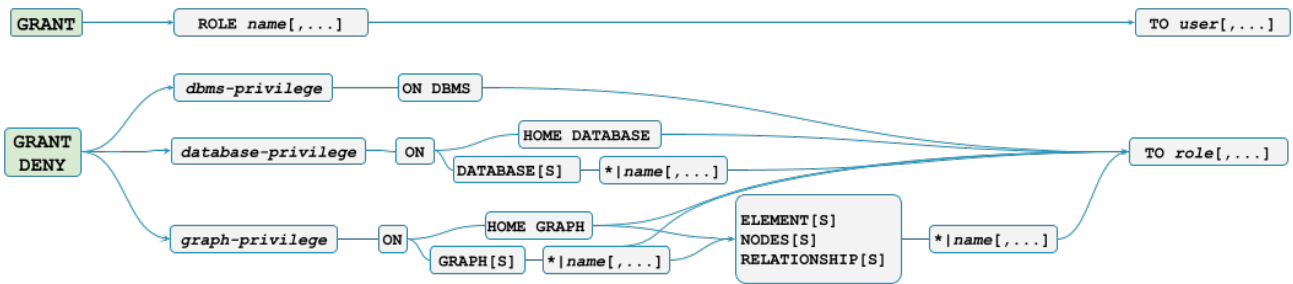


Figure 1. GRANT and DENY Syntax

A more detailed syntax illustration for graph privileges would be the following:

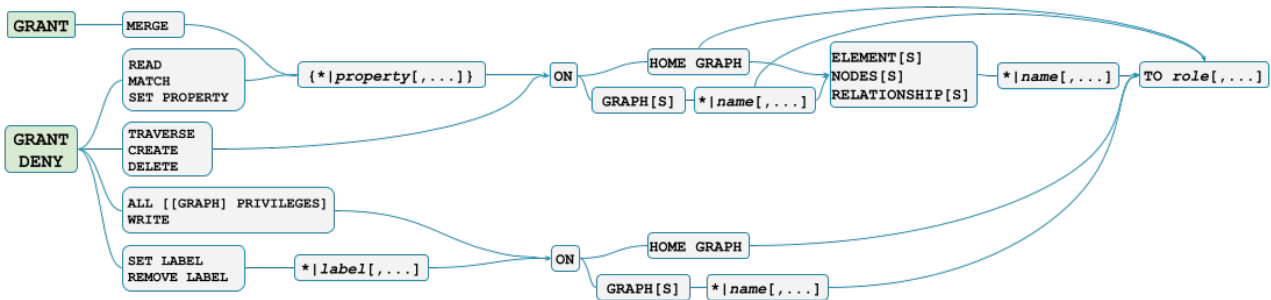


Figure 2. Syntax of GRANT and DENY Graph Privileges. The { and } are part of the syntax and not used for grouping.

The following image shows the hierarchy between different graph privileges:



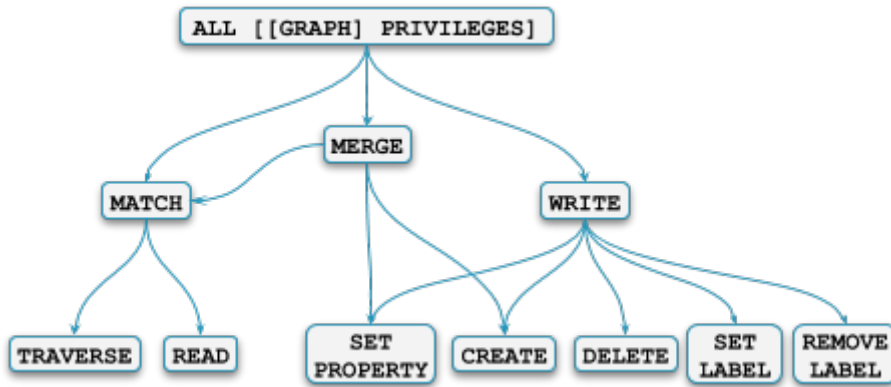


Figure 3. Graph privileges hierarchy

## Listing privileges Enterprise edition

Available privileges can be displayed using the different `SHOW PRIVILEGE[S]` commands.

Table 555. Show privileges command syntax

Command	<code>SHOW PRIVILEGE</code>
Syntax	<pre>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]]</pre>
Description	List all privileges.

Table 556. Show role privileges syntax

Command	<code>SHOW ROLE ... PRIVILEGE</code>
Syntax	<pre>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]]</pre>
Description	Lists privileges for a specific role.

Table 557. Show user privileges syntax

Command	<code>SHOW USER ... PRIVILEGE</code>
Syntax	<pre>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]]</pre>

Description	<p>Lists privileges for a specific user, or the current user.</p> <p>[NOTE] ===== 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, <code>SHOW USER PRIVILEGES</code> will only work in a limited capacity.</p> <p>Other users' privileges cannot be listed when using a non-native auth provider. =====</p>
-------------	--

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 Enterprise edition

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 558. Result

access	action	resource	graph	segment	role
"GRANTED"	"execute"	"database"	"*"	"FUNCTION(*)"	"PUBLIC"
"GRANTED"	"execute"	"database"	"*"	"PROCEDURE(*)"	"PUBLIC"
"GRANTED"	"access"	"database"	"DEFAULT"	"database"	"PUBLIC"

access	action	resource	graph	segment	role
"GRANTED"	"match"	"all_properties"	"*"	"NODE(*)"	"admin"
"GRANTED"	"write"	"graph"	"*"	"NODE(*)"	"admin"
"GRANTED"	"match"	"all_properties"	"*"	"RELATIONSHIP(*)"	"admin"
"GRANTED"	"write"	"graph"	"*"	"RELATIONSHIP(*)"	"admin"
"GRANTED"	"transaction_management"	"database"	"*"	"USER(*)"	"admin"
"GRANTED"	"access"	"database"	"*"	"database"	"admin"
"GRANTED"	"constraint"	"database"	"*"	"database"	"admin"
"GRANTED"	"dbms_actions"	"database"	"*"	"database"	"admin"
"GRANTED"	"index"	"database"	"*"	"database"	"admin"
"GRANTED"	"start_database"	"database"	"*"	"database"	"admin"
"GRANTED"	"stop_database"	"database"	"*"	"database"	"admin"
"GRANTED"	"token"	"database"	"*"	"database"	"admin"
"GRANTED"	"match"	"all_properties"	"*"	"NODE(*)"	"architect"
"GRANTED"	"write"	"graph"	"*"	"NODE(*)"	"architect"
"GRANTED"	"match"	"all_properties"	"*"	"RELATIONSHIP(*)"	"architect"
"GRANTED"	"write"	"graph"	"*"	"RELATIONSHIP(*)"	"architect"
"GRANTED"	"access"	"database"	"*"	"database"	"architect"
"GRANTED"	"constraint"	"database"	"*"	"database"	"architect"
"GRANTED"	"index"	"database"	"*"	"database"	"architect"
"GRANTED"	"token"	"database"	"*"	"database"	"architect"
"GRANTED"	"match"	"all_properties"	"*"	"NODE(*)"	"editor"
"GRANTED"	"write"	"graph"	"*"	"NODE(*)"	"editor"
"GRANTED"	"match"	"all_properties"	"*"	"RELATIONSHIP(*)"	"editor"
"GRANTED"	"write"	"graph"	"*"	"RELATIONSHIP(*)"	"editor"
"GRANTED"	"access"	"database"	"*"	"database"	"editor"
"DENIED"	"access"	"database"	"neo4j"	"database"	"noAccessUsers"
"GRANTED"	"match"	"all_properties"	"*"	"NODE(*)"	"publisher"
"GRANTED"	"write"	"graph"	"*"	"NODE(*)"	"publisher"
"GRANTED"	"match"	"all_properties"	"*"	"RELATIONSHIP(*)"	"publisher"
"GRANTED"	"write"	"graph"	"*"	"RELATIONSHIP(*)"	"publisher"
"GRANTED"	"access"	"database"	"*"	"database"	"publisher"
"GRANTED"	"token"	"database"	"*"	"database"	"publisher"
"GRANTED"	"match"	"all_properties"	"*"	"NODE(*)"	"reader"
"GRANTED"	"match"	"all_properties"	"*"	"RELATIONSHIP(*)"	"reader"
"GRANTED"	"access"	"database"	"*"	"database"	"reader"
"GRANTED"	"access"	"database"	"neo4j"	"database"	"regularUsers"

access	action	resource	graph	segment	role
Rows: 39					

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 559. Result

role	access	action	segment
"admin"	"GRANTED"	"access"	"database"
"admin"	"GRANTED"	"constraint"	"database"
"admin"	"GRANTED"	"dbms_actions"	"database"
"admin"	"GRANTED"	"index"	"database"
"admin"	"GRANTED"	"match"	"NODE(*)"
"admin"	"GRANTED"	"match"	"RELATIONSHIP(*)"
"admin"	"GRANTED"	"start_database"	"database"
"admin"	"GRANTED"	"stop_database"	"database"
"admin"	"GRANTED"	"token"	"database"
"admin"	"GRANTED"	"transaction_management"	"USER(*)"
"admin"	"GRANTED"	"write"	"NODE(*)"
"admin"	"GRANTED"	"write"	"RELATIONSHIP(*)"
Rows: 12			

**WHERE** can also be used without **YIELD**:

```
SHOW PRIVILEGES
WHERE graph <> '*'
```

In this example, the **WHERE** clause is used to filter privileges down to those that target specific graphs only.

Table 560. Result

access	action	graph	resource	role	segment
"GRANTED"	"access"	"DEFAULT"	"database"	"PUBLIC"	"database"
"DENIED"	"access"	"neo4j"	"database"	"noAccessUsers"	"database"
"GRANTED"	"access"	"neo4j"	"database"	"regularUsers"	"database"

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 561. Result

role	access	privileges
"PUBLIC"	"GRANTED"	[["*", "database", "FUNCTION(*)", "execute"], ["*", "database", "PROCEDURE(*)", "execute"], ["DEFAULT", "database", "database", "access"]]
"admin"	"GRANTED"	[["*", "all_properties", "NODE(*)", "match"], ["*", "graph", "NODE(*)", "write"], ["*", "all_properties", "RELATIONSHIP(*)", "match"], ["*", "graph", "RELATIONSHIP(*)", "write"], ["*", "database", "USER(*)", "transaction_management"], ["*", "database", "database", "access"], ["*", "database", "database", "constraint"], ["*", "database", "database", "dbms_actions"], ["*", "database", "database", "index"], ["*", "database", "database", "start_database"], ["*", "database", "database", "stop_database"], ["*", "database", "database", "token"]]
"architect"	"GRANTED"	[["*", "all_properties", "NODE(*)", "match"], ["*", "graph", "NODE(*)", "write"], ["*", "all_properties", "RELATIONSHIP(*)", "match"], ["*", "graph", "RELATIONSHIP(*)", "write"], ["*", "database", "database", "access"], ["*", "database", "database", "constraint"], ["*", "database", "database", "index"], ["*", "database", "database", "token"]]
"editor"	"GRANTED"	[["*", "all_properties", "NODE(*)", "match"], ["*", "graph", "NODE(*)", "write"], ["*", "all_properties", "RELATIONSHIP(*)", "match"], ["*", "graph", "RELATIONSHIP(*)", "write"], ["*", "database", "database", "access"]]
"noAccessUsers"	"DENIED"	[["neo4j", "database", "database", "access"]]
"publisher"	"GRANTED"	[["*", "all_properties", "NODE(*)", "match"], ["*", "graph", "NODE(*)", "write"], ["*", "all_properties", "RELATIONSHIP(*)", "match"], ["*", "graph", "RELATIONSHIP(*)", "write"], ["*", "database", "database", "access"], ["*", "database", "database", "token"]]
"reader"	"GRANTED"	[["*", "all_properties", "NODE(*)", "match"], ["*", "all_properties", "RELATIONSHIP(*)", "match"], ["*", "database", "database", "access"]]
"regularUsers"	"GRANTED"	[["neo4j", "database", "database", "access"]]

Rows: 8

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.

```
SHOW PRIVILEGES YIELD * RETURN * ORDER BY role SKIP 5 LIMIT 5
```

Table 562. Result

access	action	graph	resource	role	segment
"GRANTED"	"match"	"*"	"all_properties"	"admin"	"RELATIONSHIP(*)"
"GRANTED"	"write"	"*"	"graph"	"admin"	"RELATIONSHIP(*)"
"GRANTED"	"transaction_management"	"*"	"database"	"admin"	"USER(*)"
"GRANTED"	"access"	"*"	"database"	"admin"	"database"
"GRANTED"	"constraint"	"*"	"database"	"admin"	"database"

Rows: 5

Available privileges can also be displayed as Cypher commands by adding `AS COMMAND[S]`:

```
SHOW PRIVILEGES AS COMMANDS
```

Table 563. Result

command
"DENY ACCESS ON DATABASE neo4j TO `noAccessUsers`"
"GRANT ACCESS ON DATABASE * TO `admin`"
"GRANT ACCESS ON DATABASE * TO `architect`"
"GRANT ACCESS ON DATABASE * TO `editor`"
"GRANT ACCESS ON DATABASE * TO `publisher`"
"GRANT ACCESS ON DATABASE * TO `reader`"
"GRANT ACCESS ON DATABASE neo4j TO `regularUsers`"
"GRANT ACCESS ON HOME DATABASE TO `PUBLIC`"
"GRANT ALL DBMS PRIVILEGES ON DBMS TO `admin`"
"GRANT CONSTRAINT MANAGEMENT ON DATABASE * TO `admin`"
"GRANT CONSTRAINT MANAGEMENT ON DATABASE * TO `architect`"
"GRANT EXECUTE FUNCTION * ON DBMS TO `PUBLIC`"
"GRANT EXECUTE PROCEDURE * ON DBMS TO `PUBLIC`"
"GRANT INDEX MANAGEMENT ON DATABASE * TO `admin`"
"GRANT INDEX MANAGEMENT ON DATABASE * TO `architect`"
"GRANT MATCH {*} ON GRAPH * NODE * TO `admin`"
"GRANT MATCH {*} ON GRAPH * NODE * TO `architect`"
"GRANT MATCH {*} ON GRAPH * NODE * TO `editor`"
"GRANT MATCH {*} ON GRAPH * NODE * TO `publisher`"
"GRANT MATCH {*} ON GRAPH * NODE * TO `reader`"
"GRANT MATCH {*} ON GRAPH * RELATIONSHIP * TO `admin`"
"GRANT MATCH {*} ON GRAPH * RELATIONSHIP * TO `architect`"

command
"GRANT MATCH {*} ON GRAPH * RELATIONSHIP * TO `editor`"
"GRANT MATCH {*} ON GRAPH * RELATIONSHIP * TO `publisher`"
"GRANT MATCH {*} ON GRAPH * RELATIONSHIP * TO `reader`"
"GRANT NAME MANAGEMENT ON DATABASE * TO `admin`"
"GRANT NAME MANAGEMENT ON DATABASE * TO `architect`"
"GRANT NAME MANAGEMENT ON DATABASE * TO `publisher`"
"GRANT START ON DATABASE * TO `admin`"
"GRANT STOP ON DATABASE * TO `admin`"
"GRANT TRANSACTION MANAGEMENT (*) ON DATABASE * TO `admin`"
"GRANT WRITE ON GRAPH * TO `admin`"
"GRANT WRITE ON GRAPH * TO `architect`"
"GRANT WRITE ON GRAPH * TO `editor`"
"GRANT WRITE ON GRAPH * TO `publisher`"
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 564. Result

command
"GRANT CONSTRAINT MANAGEMENT ON DATABASE * TO `admin`"
"GRANT CONSTRAINT MANAGEMENT ON DATABASE * TO `architect`"
"GRANT INDEX MANAGEMENT ON DATABASE * TO `admin`"
"GRANT INDEX MANAGEMENT ON DATABASE * TO `architect`"
"GRANT NAME MANAGEMENT ON DATABASE * TO `admin`"
"GRANT NAME MANAGEMENT ON DATABASE * TO `architect`"
"GRANT NAME MANAGEMENT ON DATABASE * TO `publisher`"
"GRANT TRANSACTION MANAGEMENT (*) ON DATABASE * TO `admin`"
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 565. Result

**command**

```
"REVOKE DENY ACCESS ON DATABASE neo4j FROM `noAccessUsers`"  
"REVOKE GRANT ACCESS ON DATABASE * FROM `admin`"  
"REVOKE GRANT ACCESS ON DATABASE * FROM `architect`"  
"REVOKE GRANT ACCESS ON DATABASE * FROM `editor`"  
"REVOKE GRANT ACCESS ON DATABASE * FROM `publisher`"  
"REVOKE GRANT ACCESS ON DATABASE * FROM `reader`"  
"REVOKE GRANT ACCESS ON DATABASE neo4j FROM `regularUsers`"  
"REVOKE GRANT ACCESS ON HOME DATABASE FROM `PUBLIC`"  
"REVOKE GRANT ALL DBMS PRIVILEGES ON DBMS FROM `admin`"  
"REVOKE GRANT CONSTRAINT MANAGEMENT ON DATABASE * FROM `admin`"  
"REVOKE GRANT CONSTRAINT MANAGEMENT ON DATABASE * FROM `architect`"  
"REVOKE GRANT EXECUTE FUNCTION * ON DBMS FROM `PUBLIC`"  
"REVOKE GRANT EXECUTE PROCEDURE * ON DBMS FROM `PUBLIC`"  
"REVOKE GRANT INDEX MANAGEMENT ON DATABASE * FROM `admin`"  
"REVOKE GRANT INDEX MANAGEMENT ON DATABASE * FROM `architect`"  
"REVOKE GRANT MATCH {*} ON GRAPH * NODE * FROM `admin`"  
"REVOKE GRANT MATCH {*} ON GRAPH * NODE * FROM `architect`"  
"REVOKE GRANT MATCH {*} ON GRAPH * NODE * FROM `editor`"  
"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]]
  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 566. Result

access	action	graph	resource	role	segment
"GRANTED"	"access"	"database"	"neo4j"	"database"	"regularUsers"

Rows: 1

```
SHOW ROLES regularUsers, noAccessUsers PRIVILEGES
```

Lists all privileges for roles `regularUsers` and `noAccessUsers`.

Table 567. Result

access	action	graph	resource	role	segment
"DENIED"	"access"	"database"	"neo4j"	"database"	"noAccessUsers"
"GRANTED"	"access"	"database"	"neo4j"	"database"	"regularUsers"

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 568. Result

command
"GRANT ACCESS ON DATABASE * TO `admin`"
"GRANT ALL DBMS PRIVILEGES ON DBMS TO `admin`"
"GRANT CONSTRAINT MANAGEMENT ON DATABASE * TO `admin`"
"GRANT INDEX MANAGEMENT ON DATABASE * TO `admin`"

command
"GRANT MATCH {*} ON GRAPH * NODE * TO `admin`"
"GRANT MATCH {*} ON GRAPH * RELATIONSHIP * TO `admin`"
"GRANT NAME MANAGEMENT ON DATABASE * TO `admin`"
"GRANT START ON DATABASE * TO `admin`"
"GRANT STOP ON DATABASE * TO `admin`"
"GRANT TRANSACTION MANAGEMENT (*) ON DATABASE * TO `admin`"
"GRANT WRITE ON GRAPH * TO `admin`"
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 569. Result

command
"GRANT MATCH {*} ON GRAPH * NODE * TO `architect`"
"GRANT MATCH {*} ON GRAPH * RELATIONSHIP * TO `architect`"
Rows: 2

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 570. 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`.



Note that if a non-native auth provider like LDAP is in use, `SHOW USER PRIVILEGES` will only work with a limited capacity as it is only possible for a user to show their own privileges. Other users' privileges cannot be listed when using a non-native auth provider.

```
SHOW USER[S] [name[, ...]] PRIVILEGE[S] [AS [REVOKE] COMMAND[S]]
  [WHERE expression]

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]
```

```
SHOW USER jake PRIVILEGES
```

Lists all privileges for user `jake`.

Table 571. Result

access	action	resource	graph	resource	role	segment
"GRANTED"	"execute"	"database"	"*"	"FUNCTION(*)"	"PUBLIC"	"jake"
"GRANTED"	"execute"	"database"	"*"	"PROCEDURE(*)"	"PUBLIC"	"jake"
"GRANTED"	"access"	"database"	"DEFAULT"	"database"	"PUBLIC"	"jake"
"GRANTED"	"access"	"database"	"neo4j"	"database"	"regularUsers"	"jake"

Rows: 4

```
SHOW USERS jake, joe PRIVILEGES
```

Lists all privileges for users `jake` and `joe`.

Table 572. Result

access	action	resource	graph	resource	role	segment
"GRANTED"	"execute"	"database"	"*"	"FUNCTION(*)"	"PUBLIC"	"jake"
"GRANTED"	"execute"	"database"	"*"	"PROCEDURE(*)"	"PUBLIC"	"jake"
"GRANTED"	"access"	"database"	"DEFAULT"	"database"	"PUBLIC"	"jake"
"GRANTED"	"access"	"database"	"neo4j"	"database"	"regularUsers"	"jake"
"GRANTED"	"execute"	"database"	"*"	"FUNCTION(*)"	"PUBLIC"	"joe"
"GRANTED"	"execute"	"database"	"*"	"PROCEDURE(*)"	"PUBLIC"	"joe"
"GRANTED"	"access"	"database"	"DEFAULT"	"database"	"PUBLIC"	"joe"
"DENIED"	"access"	"database"	"neo4j"	"database"	"noAccessUsers"	"joe"

Rows: 8

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 573. Result

command
"GRANT ACCESS ON DATABASE neo4j TO \$role"
"GRANT ACCESS ON HOME DATABASE TO \$role"
"GRANT EXECUTE FUNCTION * ON DBMS TO \$role"
"GRANT EXECUTE PROCEDURE * ON DBMS TO \$role"

Rows: 4

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 574. Result

command
"REVOKE GRANT EXECUTE FUNCTION * ON DBMS FROM \$role"
"REVOKE GRANT EXECUTE PROCEDURE * ON DBMS FROM \$role"

Rows: 2

## 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
```

## Built-in roles and privileges

*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](#)

### 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 575. Result

command
"GRANT ACCESS ON HOME DATABASE TO `PUBLIC`"

command
"GRANT EXECUTE FUNCTION * ON DBMS TO `PUBLIC`"
"GRANT EXECUTE PROCEDURE * ON DBMS TO `PUBLIC`"
Rows: 3

## 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.

## 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 576. Result

command
"GRANT ACCESS ON DATABASE * TO `reader`"
"GRANT MATCH {*} ON GRAPH * NODE * TO `reader`"
"GRANT MATCH {*} ON GRAPH * RELATIONSHIP * TO `reader`"
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 * TO reader
```

The resulting `reader` role now has the same privileges as the original built-in `reader` role.

## 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 577. Result

command
"GRANT ACCESS ON DATABASE * TO `editor`"
"GRANT MATCH {*} ON GRAPH * NODE * TO `editor`"
"GRANT MATCH {*} ON GRAPH * RELATIONSHIP * TO `editor`"
"GRANT WRITE ON GRAPH * TO `editor`"
Rows: 4

## 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 WRITE ON GRAPH * TO editor
```

The resulting `editor` role now has the same privileges as the original built-in `editor` role.

## 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 578. Result

command
"GRANT ACCESS ON DATABASE * TO `publisher`"
"GRANT MATCH {*} ON GRAPH * NODE * TO `publisher`"
"GRANT MATCH {*} ON GRAPH * RELATIONSHIP * TO `publisher`"
"GRANT NAME MANAGEMENT ON DATABASE * TO `publisher`"
"GRANT WRITE ON GRAPH * TO `publisher`"
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.



## 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 579. Result

command
"GRANT ACCESS ON DATABASE * TO `architect`"
"GRANT CONSTRAINT MANAGEMENT ON DATABASE * TO `architect`"
"GRANT INDEX MANAGEMENT ON DATABASE * TO `architect`"
"GRANT MATCH {*} ON GRAPH * NODE * TO `architect`"
"GRANT MATCH {*} ON GRAPH * RELATIONSHIP * TO `architect`"
"GRANT NAME MANAGEMENT ON DATABASE * TO `architect`"
"GRANT WRITE ON GRAPH * TO `architect`"

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:

```
GRANT ACCESS ON DATABASE * TO architect
```

```
GRANT MATCH {*} ON GRAPH * TO architect
```

```
GRANT WRITE ON GRAPH * TO architect
```

```
GRANT NAME MANAGEMENT ON DATABASE * TO architect
```

```
GRANT INDEX MANAGEMENT ON DATABASE * TO architect
```

```
GRANT CONSTRAINT MANAGEMENT ON DATABASE * TO architect
```

The resulting `architect` role now has the same privileges as the original built-in `architect` role.

## 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

```
SHOW ROLE admin PRIVILEGES AS COMMANDS
```

Table 580. Result

command
"GRANT ACCESS ON DATABASE * TO `admin`"
"GRANT ALL DBMS PRIVILEGES ON DBMS TO `admin`"
"GRANT CONSTRAINT MANAGEMENT ON DATABASE * TO `admin`"
"GRANT INDEX MANAGEMENT ON DATABASE * TO `admin`"
"GRANT MATCH {*} ON GRAPH * NODE * TO `admin`"
"GRANT MATCH {*} ON GRAPH * RELATIONSHIP * TO `admin`"
"GRANT NAME MANAGEMENT ON DATABASE * TO `admin`"
"GRANT START ON DATABASE * TO `admin`"
"GRANT STOP ON DATABASE * TO `admin`"
"GRANT TRANSACTION MANAGEMENT (*) ON DATABASE * TO `admin`"
"GRANT WRITE ON GRAPH * TO `admin`"

```
command
```

```
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:

```
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](#).

## 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.



More details about the syntax descriptions can be found [here](#).

## The **TRAVERSE** privilege

Users can be granted the right to find nodes and relationships using the **GRANT TRAVERSE** privilege.

```
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**:

```
GRANT TRAVERSE ON GRAPH neo4j NODES Post TO regularUsers
```

The **TRAVERSE** privilege can also be denied.

```
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**:

```
DENY TRAVERSE ON HOME GRAPH NODES Payments TO regularUsers
```

## 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.

```
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.

```
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.

```
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:

```
DENY READ { secret } ON GRAPH neo4j NODES Post TO regularUsers
```

## 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.

```
GRANT MATCH "{" { * | property[, ...] } }"
ON { HOME GRAPH | GRAPH[S] { * | name[, ...] } }
[
  ELEMENT[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:

```
GRANT MATCH { language, length } ON GRAPH neo4j NODES Message TO regularUsers
```

Like all other privileges, the **MATCH** privilege can also be denied.

```
DENY MATCH "{ * | property[, ...] }"
ON { HOME GRAPH | GRAPH[S] { * | name[, ...] } }
[
  ELEMENT[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).

```
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**:

```
DENY MATCH { * } ON GRAPH neo4j NODES Account TO regularUsers
```

## 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:

- **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.



More details about the syntax descriptions can be found [here](#).

## 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.

## The **DELETE** privilege

The **DELETE** privilege allows a user to delete node and relationship elements on a graph. See the Cypher **DELETE** clause.

```
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:

```
GRANT DELETE ON GRAPH neo4j ELEMENTS * TO regularUsers
```

The **DELETE** privilege can also be denied:

```
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:

```
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.

## The **SET LABEL** privilege

The **SET LABEL** privilege allows you to set labels on a node using the **SET** clause:

```
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:

```
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:

```
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:

```
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.

## The **REMOVE LABEL** privilege

The **REMOVE LABEL** privilege allows you to remove labels from a node by using the [REMOVE clause](#):

```
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:

```
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:

```
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:

```
DENY REMOVE LABEL foo ON GRAPH * TO regularUsers
```

## 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.

## 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.

## The **WRITE** privilege

The **WRITE** privilege allows the user to execute any **WRITE** command on a graph.

```
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:

```
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:

```
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:

```
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.

## The `ALL GRAPH PRIVILEGES` privilege

The `ALL GRAPH PRIVILEGES` privilege allows the user to execute any command on a graph:

```
source, syntax, role="noheader"]
```

```
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
```

## 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:


- **command:**
  - **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 new node labels to be created.
  - **CREATE NEW [RELATIONSHIP] TYPE** - allows new relationship types to be created.
  - **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.

	If you delete a database and create a new one with the same name, the new one will NOT have the same privileges previously assigned to the deleted one.
---	---

- The name component can be **\***, which means all databases. Databases created after this command execution will also be associated with these privileges.
- The **DATABASE[S] name** part of the command can be replaced by **HOME DATABASE**. This refers to the home database configured for a user or, if that user does not have a home database configured, the default database. If the user's home database changes for any reason after this command execution, the new one will be associated with these privileges. This can be quite powerful as it allows permissions to be switched from one database to another simply by changing a user's home database.

- role[, ...]

- The role or roles to associate the privilege with, comma-separated.

	More details about the syntax descriptions can be found <a href="#">here</a> .
---	--

Table 581. General grant ON DATABASE privilege syntax

Command	<b>GRANT ... ON ... TO ...</b>
Syntax	<pre>GRANT database-privilege ON { HOME DATABASE   DATABASE[S] { *   name[, ...] } } TO role[, ...]</pre>
Description	Grants a privilege to one or multiple roles.

Table 582. General deny ON DATABASE privilege syntax

Command	<b>DENY ... ON ... TO ...</b>
Syntax	<pre>DENY database-privilege ON { HOME DATABASE   DATABASE[S] { *   name[, ...] } } TO role[, ...]</pre>
Description	Denies a privilege to one or multiple roles.

Table 583. General revoke ON DATABASE privilege syntax

Command	REVOKE GRANT ... ON ... FROM ...
Syntax	REVOKE GRANT database-privilege ON { HOME DATABASE   DATABASE[S] { *   name[, ...] } } FROM role[, ...]
Description	Revoke a granted privilege from one or multiple roles.

Table 584. General revoke ON DATABASE privilege syntax

Command	REVOKE DENY ... ON ... FROM ...
Syntax	REVOKE DENY database-privilege ON { HOME DATABASE   DATABASE[S] { *   name[, ...] } } FROM role[, ...]
Description	Revokes a denied privilege from one or multiple roles.

Table 585. General revoke ON DATABASE privilege syntax

Command	REVOKE ... ON ... FROM ...
Syntax	REVOKE database-privilege ON { HOME DATABASE   DATABASE[S] { *   name[, ...] } } FROM role[, ...]
Description	Revokes a granted or denied privilege from one or multiple roles.



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.

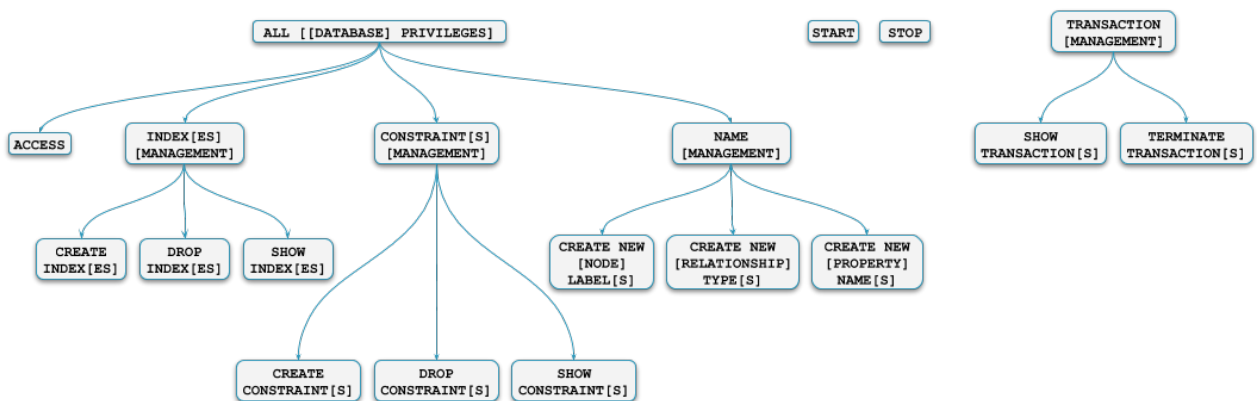


Figure 4. Database privileges hierarchy

Table 586. Database privilege syntax

Command	GRANT ACCESS
---------	--------------

Syntax	<pre>GRANT ACCESS   ON { HOME DATABASE   DATABASE[S] { *   name[, ...] } }   TO role[, ...]</pre>
Description	<p>Grants the specified roles the privilege to access:</p> <ul style="list-style-type: none"> <li>• The home database.</li> <li>• Specific database(s) or remote database alias(es).</li> <li>• All databases and remote database aliases.</li> </ul>

Table 587. Database privilege syntax

Command	<code>GRANT { START   STOP }</code>
Syntax	<pre>GRANT { START   STOP }   ON { HOME DATABASE   DATABASE[S] { *   name[, ...] } }   TO role[, ...]</pre>
Description	Grants the specified roles the privilege to start or stop the home database, specific database(s), or all databases.

Table 588. Database privilege syntax

Command	<code>GRANT { CREATE   DROP   SHOW } INDEX</code>
Syntax	<pre>GRANT { CREATE   DROP   SHOW } INDEX[ES]   ON { HOME DATABASE   DATABASE[S] { *   name[, ...] } }   TO role[, ...]</pre>
Description	Grants the specified roles the privilege to create, delete, or show indexes on the home database, specific database(s), or all databases.

Table 589. Database privilege syntax

Command	<code>GRANT INDEX</code>
Syntax	<pre>GRANT INDEX[ES] [MANAGEMENT]   ON { HOME DATABASE   DATABASE[S] { *   name[, ...] } }   TO role[, ...]</pre>
Description	Grants the specified roles the privilege to manage indexes on the home database, specific database(s), or all databases.

Table 590. Database privilege syntax

Command	<code>GRANT { CREATE   DROP   SHOW } CONSTRAINT</code>
---------	--

<b>Syntax</b>	<pre>GRANT { CREATE   DROP   SHOW } CONSTRAINT[S]       ON { HOME DATABASE   DATABASE[S] { *   name[, ...] } }       TO role[, ...]</pre>
<b>Description</b>	Grants the specified roles the privilege to create, delete, or show constraints on the home database, specific database(s), or all databases.

Table 591. Database privilege syntax

<b>Command</b>	<b>GRANT CONSTRAINT</b>
<b>Syntax</b>	<pre>GRANT CONSTRAINT[S] [MANAGEMENT]       ON { HOME DATABASE   DATABASE[S] { *   name[, ...] } }       TO role[, ...]</pre>
<b>Description</b>	Grants the specified roles the privilege to manage constraints on the home database, specific database(s), or all databases.

Table 592. Database privilege syntax

<b>Command</b>	<b>GRANT CREATE NEW LABEL</b>
<b>Syntax</b>	<pre>GRANT CREATE NEW [NODE] LABEL[S]       ON { HOME DATABASE   DATABASE[S] { *   name[, ...] } }       TO role[, ...]</pre>
<b>Description</b>	Grants the specified roles the privilege to create new node labels in the home database, specific database(s), or all databases.

Table 593. Database privilege syntax

<b>Command</b>	<b>GRANT CREATE NEW TYPE</b>
<b>Syntax</b>	<pre>GRANT CREATE NEW [RELATIONSHIP] TYPE[S]       ON { HOME DATABASE   DATABASE[S] { *   name[, ...] } }       TO role[, ...]</pre>
<b>Description</b>	Grants the specified roles the privilege to create new relationship types in the home database, specific database(s), or all databases.

Table 594. Database privilege syntax

<b>Command</b>	<b>GRANT CREATE NEW NAME</b>
<b>Syntax</b>	<pre>GRANT CREATE NEW [PROPERTY] NAME[S]       ON { HOME DATABASE   DATABASE[S] { *   name[, ...] } }       TO role[, ...]</pre>
<b>Description</b>	Grants the specified roles the privilege to create new property names in the home database, specific database(s), or all databases.



Table 595. Database privilege syntax

Command	GRANT NAME
Syntax	<pre>GRANT NAME [MANAGEMENT] ON { HOME DATABASE   DATABASE[S] { *   name[, ...] } } TO role[, ...]</pre>
Description	Grants the specified roles the privilege to manage new labels, relationship types, and property names in the home database, specific database(s), or all databases.

Table 596. Database privilege syntax

Command	GRANT ALL
Syntax	<pre>GRANT ALL [[DATABASE] PRIVILEGES] ON { HOME DATABASE   DATABASE[S] { *   name[, ...] } } TO role[, ...]</pre>
Description	Grants the specified roles all privileges for the home, a specific, or all databases and remote database aliases.

Table 597. Database privilege syntax

Command	GRANT { SHOW   TERMINATE } TRANSACTION
Syntax	<pre>GRANT { SHOW   TERMINATE } TRANSACTION[S] [( { *   user[, ...] } )] ON { HOME DATABASE   DATABASE[S] { *   name[, ...] } } TO role[, ...]</pre>
Description	Grants the specified roles the privilege to list and end the transactions and queries of all users or a particular user(s) in the home database, specific database(s), or all databases.

Table 598. Database privilege syntax

Command	GRANT TRANSACTION
Syntax	<pre>GRANT TRANSACTION [MANAGEMENT] [( { *   user[, ...] } )] ON { HOME DATABASE   DATABASE[S] { *   name[, ...] } } TO role[, ...]</pre>
Description	Grants the specified roles the privilege to manage the transactions and queries of all users or a particular user(s) in the home database, specific database(s), or all databases.

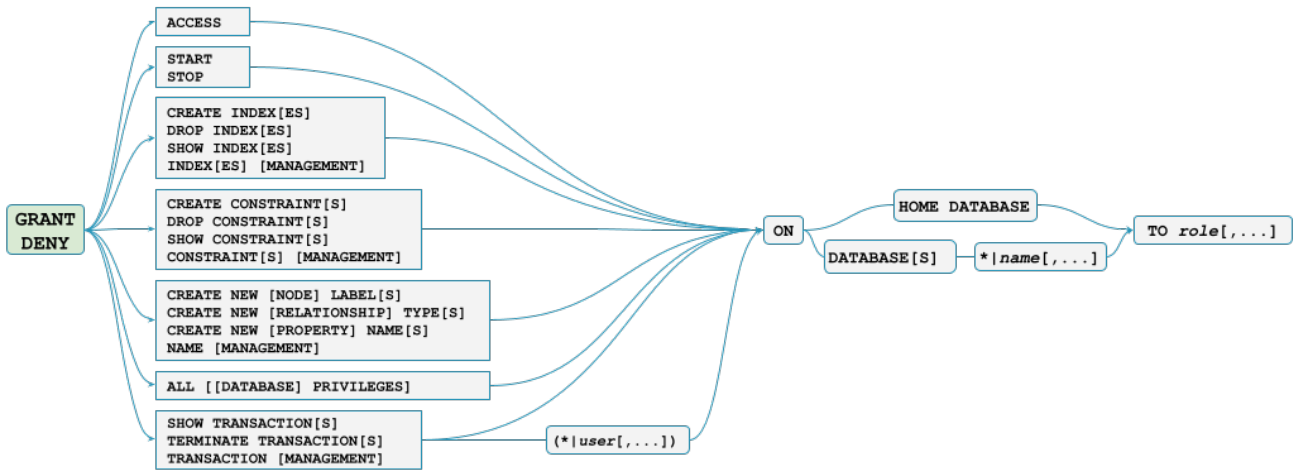


Figure 5. Syntax of GRANT and DENY Database Privileges

## 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 to 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 599. Result

command
"DENY ACCESS ON DATABASE remote-db TO `regularUsers`"
"GRANT ACCESS ON DATABASE neo4j TO `regularUsers`"

```
command
```

```
Rows: 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:

```
DENY START
ON { HOME DATABASE | DATABASE[S] { * | name[, ...] } }
TO role[, ...]
```

For example, to deny the role **regularUsers** the ability to start to 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 { HOME DATABASE | DATABASE[S] { * | name[, ...] } }
TO role[, ...]
```

For example, to grant the role **regularUsers** the ability to stop the database **neo4j**, use:

```
GRANT STOP ON DATABASE neo4j TO regularUsers
```

The **STOP** privilege can also be denied:

```
DENY STOP
ON { HOME DATABASE | DATABASE[S] { * | name[, ...] } }
TO role[, ...]
```

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 600. Result

command
"DENY ACCESS ON DATABASE remote-db TO `regularUsers`"
"DENY START ON DATABASE system TO `regularUsers`"
"DENY STOP ON DATABASE system TO `regularUsers`"
"GRANT ACCESS ON DATABASE neo4j TO `regularUsers`"
"GRANT START ON DATABASE neo4j TO `regularUsers`"
"GRANT STOP ON DATABASE neo4j TO `regularUsers`"
Rows: 6



Note that **START** and **STOP** privileges are not included in the **ALL DATABASE PRIVILEGES**.

## 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 601. Index management privilege syntax

Command	<b>GRANT { CREATE   DROP   SHOW } INDEX</b>
Syntax	<pre>GRANT { CREATE   DROP   SHOW } INDEX[ES]   ON { HOME DATABASE   DATABASE[S] { *   name[, ...] } }   TO role[, ...]</pre>
Description	Enables the specified roles to create, delete, or show indexes in the home database, specific database(s), or all databases.

Table 602. Index management privilege syntax

Command	<b>GRANT INDEX</b>
Syntax	<pre>GRANT INDEX[ES] [MANAGEMENT]   ON { HOME DATABASE   DATABASE[S] { *   name[, ...] } }   TO role[, ...]</pre>
Description	Enables the specified roles to manage indexes in the home database, specific database(s), or all databases.

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`.

## 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`, `GRANT SHOW CONSTRAINT` commands. The privilege to do all three can be granted with `GRANT CONSTRAINT MANAGEMENT` command.

Table 603. Constraint management privilege syntax

Command	<code>GRANT { CREATE   DROP   SHOW } CONSTRAINT</code>
Syntax	<pre>GRANT { CREATE   DROP   SHOW } CONSTRAINT[S] ON { HOME DATABASE   DATABASE[S] { *   name[, ...] } } TO role[, ...]</pre>
Description	Enables the specified roles to create, delete, or show constraints on the home database, specific database(s), or all databases.

Table 604. Constraint management privilege syntax

Command	<code>GRANT CONSTRAINT</code>
Syntax	<pre>GRANT CONSTRAINT[S] [MANAGEMENT] ON { HOME DATABASE   DATABASE[S] { *   name[, ...] } } TO role[, ...]</pre>
Description	Enable the specified roles to manage constraints on the home database, specific database(s), or all databases.

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
```

The `SHOW CONSTRAINTS` privilege only affects the `SHOW CONSTRAINTS` command, and not the older procedures for listing constraints, such as `db.constraints`.

## 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 605. Node label management privileges syntax

Command	<code>GRANT CREATE NEW LABEL</code>
---------	-------------------------------------

Syntax	<pre>GRANT CREATE NEW [NODE] LABEL[S]   ON { HOME DATABASE   DATABASE[S] { *   name[, ...] } }   TO role[, ...]</pre>
Description	Enables the specified roles to create new node labels in the home database, specific database(s), or all databases.

Table 606. Relationship type management privileges syntax

Command	GRANT CREATE NEW TYPE
Syntax	<pre>GRANT CREATE NEW [RELATIONSHIP] TYPE[S]   ON { HOME DATABASE   DATABASE[S] { *   name[, ...] } }   TO role[, ...]</pre>
Description	Enables the specified roles to create new relationship types in the home database, specific database(s), or all databases.

Table 607. Property name management privileges syntax

Command	GRANT CREATE NEW NAME
Syntax	<pre>GRANT CREATE NEW [PROPERTY] NAME[S]   ON { HOME DATABASE   DATABASE[S] { *   name[, ...] } }   TO role[, ...]</pre>
Description	Enables the specified roles to create new property names in the home database, specific database(s), or all databases.

Table 608. Node label, relationship type, and property name privileges management syntax

Command	GRANT NAME
Syntax	<pre>GRANT NAME [MANAGEMENT]   ON { HOME DATABASE   DATABASE[S] { *   name[, ...] } }   TO role[, ...]</pre>
Description	Enables the specified roles to create new labels, relationship types, and property names in the home database, specific database(s), or all databases.

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
```

## 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.

```
GRANT ALL DATABASE PRIVILEGES ON DATABASE neo4j TO databaseAdminUsers
```

The privileges granted can be seen using the **SHOW PRIVILEGES** command:

```
SHOW ROLE databaseAdminUsers PRIVILEGES AS COMMANDS
```

Table 609. Result

command
"GRANT ALL DATABASE PRIVILEGES ON DATABASE neo4j TO `databaseAdminUsers`"
Rows: 1

## 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 610. Database privilege syntax

Command	GRANT SHOW TRANSACTION
Syntax	<pre>GRANT SHOW TRANSACTION[S] [( { *   user[, ...] } )] ON { HOME DATABASE   DATABASE[S] { *   name[, ...] } } TO role[, ...]</pre>
Description	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.

Table 611. Database privilege syntax

Command	GRANT TERMINATE TRANSACTION
---------	-----------------------------

Syntax	<pre>GRANT TERMINATE TRANSACTION[S] [( { *   user[, ...] } )] ON { HOME DATABASE   DATABASE[S] { *   name[, ...] } } TO role[, ...]</pre>
Description	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.

Table 612. Database privilege syntax

Command	<code>GRANT TRANSACTION</code>
Syntax	<pre>GRANT TRANSACTION [MANAGEMENT] [( { *   user[, ...] } )] ON { HOME DATABASE   DATABASE[S] { *   name[, ...] } } TO role[, ...]</pre>
Description	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
```

## 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 [key Neo4j terminology](#).



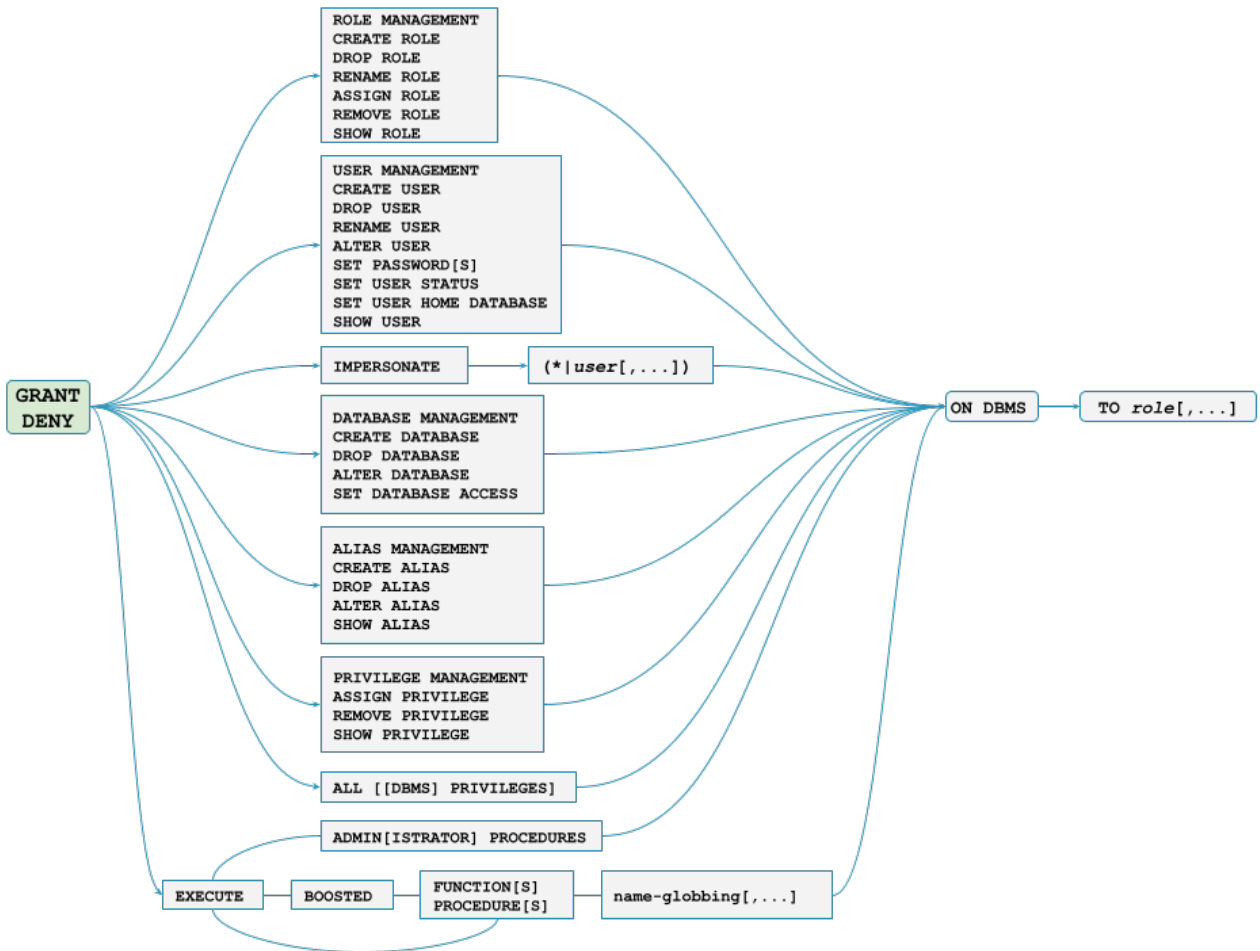


Figure 6. Syntax of GRANT and DENY DBMS Privileges

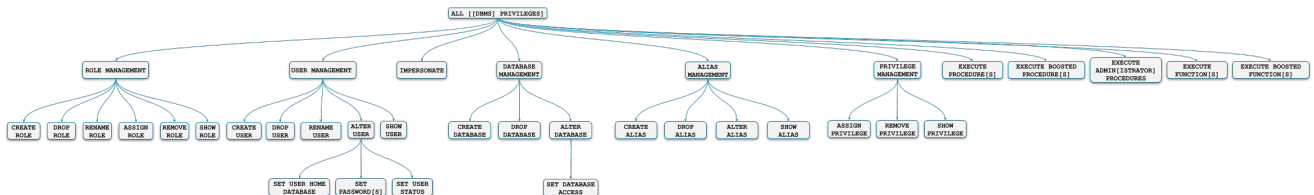


Figure 7. DBMS privileges hierarchy

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

## 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:

```
SHOW ROLE usermanager PRIVILEGES AS COMMANDS
```

Table 613. Result

command
"GRANT ROLE MANAGEMENT ON DBMS TO `usermanager`"
"GRANT USER MANAGEMENT ON DBMS TO `usermanager`"
Rows: 2

Note that this role does not 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 614. Result

command
"DENY DATABASE MANAGEMENT ON DBMS TO `customAdministrator`"
"GRANT ALL DBMS PRIVILEGES ON DBMS TO `customAdministrator`"
"GRANT TRANSACTION MANAGEMENT (*) ON DATABASE * TO `customAdministrator`"
Rows: 3

## 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.



More details about the syntax descriptions can be found [here](#).

Table 615. Role management privileges command syntax

Command	Description
<pre>GRANT CREATE ROLE ON DBMS TO role[, ...]</pre>	Enables the specified roles to create new roles.
<pre>GRANT RENAME ROLE ON DBMS TO role[, ...]</pre>	Enables the specified roles to change the name of roles.
<pre>GRANT DROP ROLE ON DBMS TO role[, ...]</pre>	Enables the specified roles to delete roles.
<pre>GRANT ASSIGN ROLE ON DBMS TO role[, ...]</pre>	Enables the specified roles to assign roles to users.
<pre>GRANT REMOVE ROLE ON DBMS TO role[, ...]</pre>	Enables the specified roles to remove roles from users.
<pre>GRANT SHOW ROLE ON DBMS TO role[, ...]</pre>	Enables the specified roles to list roles.
<pre>GRANT ROLE MANAGEMENT ON DBMS TO role[, ...]</pre>	Enables the specified roles to create, delete, assign, remove, and list roles.

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 616. Result

command
"GRANT CREATE ROLE ON DBMS TO `roleAdder`"
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 617. Result

command
"GRANT RENAME ROLE ON DBMS TO `roleNameModifier`"
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 618. Result

command
"GRANT DROP ROLE ON DBMS TO `roleDropper`"
Rows: 1

The ability to assign roles to users can be granted via the **ASSIGN ROLE** privilege. See an example:

```
GRANT ASSIGN ROLE ON DBMS TO roleAssigner
```

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 619. Result

command
"GRANT ASSIGN ROLE ON DBMS TO `roleAssigner`"

```
command
```

```
Rows: 1
```

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 620. Result

```
command
```

```
"GRANT REMOVE ROLE ON DBMS TO `roleRemover`"
```

```
Rows: 1
```

The ability to show roles can be granted via the **SHOW ROLE** privilege. A role with this privilege is allowed to execute the **SHOW ROLES** and **SHOW POPULATED ROLES** administration commands. For the **SHOW ROLES WITH USERS** and **SHOW POPULATED ROLES WITH USERS** administration commands, both this privilege and the **SHOW USER** privilege are required. The following query shows an example of how to grant the **SHOW ROLE** privilege:

In order to use **SHOW ROLES WITH USERS** and **SHOW POPULATED ROLES WITH USERS** administration commands, both the **SHOW ROLE** 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 621. Result

```
command
```

```
"GRANT SHOW ROLE ON DBMS TO `roleShower`"
```

```
Rows: 1
```

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 622. Result

command
"GRANT ROLE MANAGEMENT ON DBMS TO `roleManager`"
Rows: 1

## 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.



More details about the syntax descriptions can be found [here](#).

Table 623. User management privileges command syntax

Command	Description
<pre>GRANT CREATE USER ON DBMS TO role[, ...]</pre>	Enables the specified roles to create new users.
<pre>GRANT RENAME USER ON DBMS TO role[, ...]</pre>	Enables the specified roles to change the name of users.
<pre>GRANT ALTER USER ON DBMS TO role[, ...]</pre>	Enables the specified roles to modify users.
<pre>GRANT SET PASSWORD[S] ON DBMS TO role[, ...]</pre>	Enables the specified roles to modify users' passwords and whether those passwords must be changed upon first login.
<pre>GRANT SET USER HOME DATABASE ON DBMS TO role[, ...]</pre>	Enables the specified roles to modify users' home database.

Command	Description
<pre>GRANT SET USER STATUS ON DBMS TO role[, ...]</pre>	Enables the specified roles to modify the account status of users.
<pre>GRANT DROP USER ON DBMS TO role[, ...]</pre>	Enables the specified roles to delete users.
<pre>GRANT SHOW USER ON DBMS TO role[, ...]</pre>	Enables the specified roles to list users.
<pre>GRANT USER MANAGEMENT ON DBMS TO role[, ...]</pre>	Enables the specified roles to create, delete, modify, and list users.

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 624. Result

command
"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 625. Result

command
"GRANT RENAME USER ON DBMS TO `userNameModifier`"
Rows: 1

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 626. Result

command
"GRANT ALTER USER ON DBMS TO `userModifier`"
Rows: 1

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:

```
SHOW ROLE passwordModifier PRIVILEGES AS COMMANDS
```

Table 627. Result

command
"GRANT SET PASSWORD ON DBMS TO `passwordModifier`"
Rows: 1

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
```

Table 628. Result

command
"GRANT SET USER STATUS ON DBMS TO `statusModifier`"
Rows: 1

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
```

Table 629. Result

command
"GRANT SET USER HOME DATABASE ON DBMS TO `statusModifier`"
"GRANT SET USER STATUS ON DBMS TO `statusModifier`"
Rows: 2

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
```

Table 630. Result

command
"GRANT DROP USER ON DBMS TO `userDropper`"
Rows: 1

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 631. Result

command
"GRANT SHOW USER ON DBMS TO `userShower`"
Rows: 1

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 632. Result

command
"GRANT SHOW USER ON DBMS TO `userManager`"
Rows: 1

## 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.



More details about the syntax descriptions can be found [here](#).

Table 633. Impersonation privileges command syntax

Command	Description
<pre>GRANT IMPERSONATE [(*)] ON DBMS TO role[, ...]</pre>	Enables the specified roles to impersonate any user.
<pre>GRANT IMPERSONATE (user[, ...]) ON DBMS TO role[, ...]</pre>	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 634. Result

command
"GRANT IMPERSONATE (*) ON DBMS TO `userImpersonator`"
Rows: 1

It is also possible to deny and revoke that privilege. See an example which shows of 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
```

## 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.



More details about the syntax descriptions can be found [here](#).

Table 635. Database management privileges command syntax

Command	Description
<pre>GRANT CREATE DATABASE ON DBMS TO role[, ...]</pre>	Enables the specified roles to create new databases and aliases.
<pre>GRANT DROP DATABASE ON DBMS TO role[, ...]</pre>	Enables the specified roles to delete databases and aliases.
<pre>GRANT ALTER DATABASE ON DBMS TO role[, ...]</pre>	Enables the specified roles to modify databases and aliases.

Command	Description
<pre>GRANT SET DATABASE ACCESS ON DBMS TO role[, ...]</pre>	Enables the specified roles to modify access of databases.
<pre>GRANT DATABASE MANAGEMENT ON DBMS TO role[, ...]</pre>	Enables the specified roles to create, delete, and modify databases and aliases.

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 636. Result

command
"GRANT CREATE DATABASE ON DBMS TO `databaseAdder`"
Rows: 1

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 637. Result

command
"GRANT DROP DATABASE ON DBMS TO `databaseDropper`"
Rows: 1

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 638. 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 639. 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 640. Result

command
"GRANT DATABASE MANAGEMENT ON DBMS TO `databaseManager`"

command

Rows: 1

## 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](#).



More details about the syntax descriptions can be found [here](#).

Table 641. Alias management privileges command syntax

Command	Description
<pre>GRANT <b>CREATE</b> ALIAS <b>ON</b> DBMS TO role[, ...]</pre>	Enables the specified roles to create new aliases.
<pre>GRANT <b>DROP</b> ALIAS <b>ON</b> DBMS TO role[, ...]</pre>	Enables the specified roles to delete aliases.
<pre>GRANT <b>ALTER</b> ALIAS <b>ON</b> DBMS TO role[, ...]</pre>	Enables the specified roles to modify aliases.
<pre>GRANT <b>SHOW</b> ALIAS <b>ON</b> DBMS TO role[, ...]</pre>	Enables the specified roles to list aliases.
<pre>GRANT <b>ALIAS MANAGEMENT</b> <b>ON</b> DBMS TO role[, ...]</pre>	Enables the specified roles to list, create, delete, and modify aliases.

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 642. Result

command
"GRANT CREATE ALIAS ON DBMS TO `aliasAdder`"
Rows: 1

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 643. Result

command
"GRANT DROP ALIAS ON DBMS TO `aliasDropper`"
Rows: 1

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 644. Result

command
"GRANT ALTER ALIAS ON DBMS TO `aliasModifier`"
Rows: 1

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 645. Result

command
"GRANT SHOW ALIAS ON DBMS TO `aliasLister`"
Rows: 1

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 646. Result

command
"GRANT ALIAS MANAGEMENT ON DBMS TO `aliasManager`"
Rows: 1

## 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.



More details about the syntax descriptions can be found [here](#).

Table 647. Privilege management privileges command syntax

Command	Description
<pre>GRANT SHOW PRIVILEGE ON DBMS TO role[, ...]</pre>	Enables the specified roles to list privileges.
<pre>GRANT ASSIGN PRIVILEGE ON DBMS TO role[, ...]</pre>	Enables the specified roles to assign privileges using the <b>GRANT</b> and <b>DENY</b> commands.

Command	Description
<pre>GRANT REMOVE PRIVILEGE ON DBMS TO role[, ...]</pre>	Enables the specified roles to remove privileges using the <b>REVOKE</b> command.
<pre>GRANT PRIVILEGE MANAGEMENT ON DBMS TO role[, ...]</pre>	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 648. Result

command
"GRANT SHOW PRIVILEGE ON DBMS TO `privilegeShower`"
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:

```
SHOW ROLE privilegeAssigner PRIVILEGES AS COMMANDS
```

Table 649. Result

command
"GRANT ASSIGN PRIVILEGE ON DBMS TO `privilegeAssigner`"
Rows: 1

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:

```
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:

```
SHOW ROLE privilegeRemover PRIVILEGES AS COMMANDS
```

Table 650. Result

command
"GRANT REMOVE PRIVILEGE ON DBMS TO `privilegeRemover`"
Rows: 1

The privileges to list, assign, and remove privileges can be granted via the `PRIVILEGE MANAGEMENT` privilege. See an example:

```
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:

```
SHOW ROLE privilegeManager PRIVILEGES AS COMMANDS
```

Table 651. Result

command
"GRANT PRIVILEGE MANAGEMENT ON DBMS TO `privilegeManager`"
Rows: 1

## 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.



More details about the syntax descriptions can be found [here](#).

Table 652. Execute privileges command syntax

Command	Description
<pre>GRANT EXECUTE PROCEDURE[S] name-globbing[, ...] ON DBMS TO role[, ...]</pre>	Enables the specified roles to execute the given procedures.
<pre>GRANT EXECUTE BOOSTED PROCEDURE[S] name-globbing[, ...] ON DBMS TO role[, ...]</pre>	Enables the specified roles to execute the given procedures with elevated privileges.
<pre>GRANT EXECUTE ADMIN[ISTRATOR] PROCEDURES ON DBMS TO role[, ...]</pre>	Enables the specified roles to execute procedures annotated with @Admin. The procedures are executed with elevated privileges.
<pre>GRANT EXECUTE [USER [DEFINED]] FUNCTION[S] name-globbing[, ...] ON DBMS TO role[, ...]</pre>	Enables the specified roles to execute the given user defined functions.
<pre>GRANT EXECUTE BOOSTED [USER [DEFINED]] FUNCTION[S] name-globbing[, ...] ON DBMS TO role[, ...]</pre>	Enables the specified roles to execute the given user defined functions with elevated privileges.

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 role 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:

```
SHOW ROLE procedureExecutor PRIVILEGES AS COMMANDS
```

Table 653. Result

command
"GRANT EXECUTE PROCEDURE db.schema.* ON DBMS TO `procedureExecutor`"
Rows: 1

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`:

```
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:

```
SHOW ROLE deniedProcedureExecutor PRIVILEGES AS COMMANDS
```

Table 654. Result

command
"DENY EXECUTE PROCEDURE dbms.killTransaction* ON DBMS TO `deniedProcedureExecutor`"
"GRANT EXECUTE PROCEDURE * ON DBMS TO `deniedProcedureExecutor`"
Rows: 2

Both the `dbms.killTransaction` and the `dbms.killTransactions` procedures 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:

```
GRANT EXECUTE BOOSTED PROCEDURE db.labels, db.relationshipTypes ON DBMS TO boostedProcedureExecutor
```

Users with the role `boostedProcedureExecutor` can thus run the `db.labels` and the `db.relationshipTypes` procedures 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 the `db.labels` and the `db.relationshipTypes` procedures, 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 655. Result

command
"GRANT EXECUTE BOOSTED PROCEDURE db.labels ON DBMS TO `boostedProcedureExecutor`"
"GRANT EXECUTE BOOSTED PROCEDURE db.relationshipTypes ON DBMS TO `boostedProcedureExecutor`"
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 489. 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 the `db.labels` procedure 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 656. Result

command
"DENY EXECUTE BOOSTED PROCEDURE db.labels ON DBMS TO `deniedBoostedProcedureExecutor1`"
"GRANT EXECUTE PROCEDURE * ON DBMS TO `deniedBoostedProcedureExecutor1`"

Rows: 2

Example 490. 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 657. Result

command
"DENY EXECUTE PROCEDURE db.labels ON DBMS TO `deniedBoostedProcedureExecutor2`"
"GRANT EXECUTE BOOSTED PROCEDURE * ON DBMS TO `deniedBoostedProcedureExecutor2`"

Rows: 2



Example 491. 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 658. Result

command
"DENY EXECUTE BOOSTED PROCEDURE db.labels ON DBMS TO `deniedBoostedProcedureExecutor3`"
"GRANT EXECUTE BOOSTED PROCEDURE * ON DBMS TO `deniedBoostedProcedureExecutor3`"

Rows: 2

Example 492. Grant EXECUTE PROCEDURE and EXECUTE BOOSTED PROCEDURE and deny EXECUTE BOOSTED PROCEDURE

```
GRANT EXECUTE PROCEDURE db.labels ON DBMS TO deniedBoostedProcedureExecutor4
```

```
GRANT EXECUTE BOOSTED PROCEDURE * ON DBMS TO deniedBoostedProcedureExecutor4
```

```
DENY EXECUTE BOOSTED PROCEDURE db.labels ON DBMS TO deniedBoostedProcedureExecutor4
```

The resulting role has privileges that allow executing all procedures with elevated privileges except the `db.labels` procedure, 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:

```
SHOW ROLE deniedBoostedProcedureExecutor4 PRIVILEGES AS COMMANDS
```

Table 659. Result

command
"DENY EXECUTE BOOSTED PROCEDURE db.labels ON DBMS TO `deniedBoostedProcedureExecutor4`"
"GRANT EXECUTE BOOSTED PROCEDURE * ON DBMS TO `deniedBoostedProcedureExecutor4`"
"GRANT EXECUTE PROCEDURE db.labels ON DBMS TO `deniedBoostedProcedureExecutor4`"

Rows: 3

Example 493. 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 660. Result

command
"GRANT EXECUTE ADMIN PROCEDURES ON DBMS TO `adminProcedureExecutor`"
Rows: 1

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 role 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 494. 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 661. Result

command
"GRANT EXECUTE FUNCTION apoc.coll.* ON DBMS TO `functionExecutor`"

Rows: 1

To allow the execution of all but a few UDFs, you can grant `EXECUTE USER DEFINED FUNCTIONS *` and deny the unwanted functions.

## Example 495. Execute user-defined functions

The following queries allow the execution of all UDFs except those starting with `apoc.any.prop`:

```
GRANT EXECUTE USER DEFINED FUNCTIONS * ON DBMS TO deniedFunctionExecutor
```

```
DENY EXECUTE USER DEFINED FUNCTION apoc.any.prop* ON DBMS TO deniedFunctionExecutor
```

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:

```
SHOW ROLE deniedFunctionExecutor PRIVILEGES AS COMMANDS
```

Table 662. Result

command
"DENY EXECUTE FUNCTION apoc.any.prop* ON DBMS TO `deniedFunctionExecutor`"
"GRANT EXECUTE FUNCTION * ON DBMS TO `deniedFunctionExecutor`"

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 496. 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 663. Result

command
"GRANT EXECUTE BOOSTED FUNCTION apoc.any.properties ON DBMS TO `boostedFunctionExecutor`"
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
```



Users with the role `globbing5` can thus run procedures `mine.public.exampleProcedure`, `mine.public.exampleProcedure1` and `mine.public.exampleProcedure42`, but none of the others.

```
GRANT EXECUTE PROCEDURE `mine.public.with#*$Characters`, mine.private.`with#Spec??*$Characters` ON DBMS TO globbing6
```

Users with the role `globbing6` can thus run procedures `mine.public.with#Special$Characters` and `mine.private.with#Special$Characters`, but none of the others.



The name-globbing may be fully or partially escaped. Both `*` and `?` are interpreted as wildcards either way.

## 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.



More details about the syntax descriptions can be found [here](#).

```
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 664. Result

command
<code>"GRANT ALL DBMS PRIVILEGES ON DBMS TO `dbmsManager`"</code>
Rows: 1

## Limitations

This section lists the known limitations and implications of Neo4j's role-based access control security.

## 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 `ndy` in either property, leading to five results.
- Filter the results to exclude nodes that also have the label `:Person`, filtering out `Mandy` and `Joe`, leading to three results.

This difference in results is caused by the `OR` relationship between the two properties in the index creation.

## Denying properties

Now consider denying access on properties, like the `surname` property:

```
DENY READ {surname} ON GRAPH * TO users
```

For that, run the same queries again:

```
MATCH (n:User)
WHERE n.name CONTAINS 'ndy'
RETURN n.name
```

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:

```
MATCH (n:User)
WHERE n.name CONTAINS 'ndy' AND n.surname IS NOT NULL
RETURN n.name
```

Since the `surname` property is denied, it will appear to always be `null` and the composite index empty. Therefore, the query returns no result.

Now consider the full-text index query:

```
CALL db.index.fulltext.queryNodes("userNames", "ndy") YIELD node, score
RETURN node.name
```

The problem remains, since it is not certain whether the results provided by the index were returned due to a match on the `name` or the `surname` property. Results from the `surname` property now need to be excluded by the security rules, because they require that the user is unable to see any `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 `name` property, or also on the disallowed `surname` property. What is known is that the earlier query returned a match for `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 **Andy** and **Sandy**, which might have been expected.

## 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:

```
GRANT TRAVERSE ON GRAPH * NODES A TO custom
```

This means that only label `:A` is explicitly allow-listed. Thus, executing

```
CALL db.labels()
```

will only return label `:A`, because that is the only label for which traversal was granted.

## 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:

```
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:

```
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:

User with <b>free</b> role	User with <b>restricted</b> role
<p>The database can access the count store and retrieve the total number of nodes with the label <b>:Person</b>.</p> <p>This is a very quick operation.</p>	<p>The database cannot access the count store because it must make sure that only traversable nodes with the desired label <b>:Person</b> are counted. Due to this, each node with the <b>:Person</b> label needs to be accessed and examined to make sure that they do not have a deny-listed label, such as <b>:Customer</b>.</p> <p>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.</p>

# Deprecations, additions and compatibility

Cypher is a graph query 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.

## Version 4.4

### Deprecated features

Feature	Details
<p>Functionality <b>Deprecated</b></p> <pre>MATCH (n) RETURN n.propertyName_1, n.propertyName_2 + count(*)</pre>	<p>Implied grouping keys are deprecated. Only expressions that do not contain aggregations are still considered grouping keys.</p> <p>In expressions that contain aggregations, the leaves must be either:</p> <ul style="list-style-type: none"><li>• An aggregation.</li><li>• A literal.</li><li>• A parameter.</li><li>• A variable, ONLY IF it is either:<ol style="list-style-type: none"><li>1) A projection expression on its own (e.g. the <code>n</code> in <code>RETURN n AS myNode, n.value + count(*)</code>).</li><li>2) A local variable in the expression (e.g. the <code>x</code> in <code>RETURN n, n.prop + size([ x IN range(1, 10)   x ])</code>).</li></ol></li><li>• Property access, ONLY IF it is also a projection expression on its own (e.g. the <code>n.prop</code> in <code>RETURN n.prop, n.prop + count(*)</code>).</li><li>• Map access, ONLY IF it is also a projection expression on its own (e.g. the <code>map.prop</code> in <code>WITH {prop: 2} AS map RETURN map.prop, map.prop + count(*)</code>).</li></ul>



Feature	Details
<p><b>Syntax</b> <b>Deprecated</b></p> <pre>USING PERIODIC COMMIT ...</pre>	<p>Replaced by:</p> <pre>CALL {   ... } IN TRANSACTIONS</pre>
<p><b>Syntax</b> <b>Deprecated</b></p> <pre>CREATE (a {prop:7})-[r:R]-&gt;(b {prop: a.prop})</pre>	<p><b>CREATE</b> clauses in which a variable introduced in the pattern is also referenced from the same pattern are deprecated.</p>
<p><b>Syntax</b> <b>Deprecated</b></p> <pre>CREATE CONSTRAINT ON ... ASSERT ...</pre>	<p>Replaced by:</p> <pre>CREATE CONSTRAINT FOR ... REQUIRE ...</pre>
<p><b>Functionality</b> <b>Deprecated</b></p> <pre>CREATE BTREE INDEX ...</pre>	<p>B-tree indexes are deprecated, partially replaced for now, and will be fully replaced in 5.0 by <a href="#">future indexes</a>. In 4.4, b-tree indexes are still the correct alternative to use.</p> <p>B-tree indexes used for string queries are replaced by:</p> <pre>CREATE TEXT INDEX ...</pre> <p>B-tree indexes used for spatial queries will be replaced by:</p>
<p><b>Functionality</b> <b>Deprecated</b></p> <pre>CREATE INDEX ... OPTIONS "{" btree-option: btree-value[, ...] "}"</pre>	<pre>CREATE POINT INDEX ...</pre> <p>B-tree indexes used for general queries or property value types will be replaced by:</p> <pre>CREATE RANGE INDEX ...</pre> <p>These new indexes may be combined for multiple use cases.</p>

Feature	Details
<div data-bbox="137 159 477 208" style="border: 1px solid #0070C0; border-radius: 3px; padding: 2px; display: inline-block; margin-right: 5px;">Functionality</div> <div data-bbox="320 159 477 208" style="border: 1px solid #0070C0; border-radius: 3px; padding: 2px; display: inline-block; margin-right: 5px;">Deprecated</div> <div data-bbox="137 237 786 309" style="border: 1px solid #ccc; border-radius: 3px; padding: 5px; margin-top: 10px;"> SHOW BTREE INDEXES </div>	<p>B-tree indexes are deprecated, partially replaced for now, and will be fully replaced in 5.0 by <a href="#">future indexes</a>. In 4.4, b-tree indexes are still the correct alternative to use.</p> <p>Replaced by the new and future index types:</p> <div data-bbox="807 439 1457 510" style="border: 1px solid #ccc; border-radius: 3px; padding: 5px; margin-top: 10px;"> SHOW {POINT   RANGE   TEXT} INDEXES </div>
<div data-bbox="137 562 477 611" style="border: 1px solid #0070C0; border-radius: 3px; padding: 2px; display: inline-block; margin-right: 5px;">Functionality</div> <div data-bbox="320 562 477 611" style="border: 1px solid #0070C0; border-radius: 3px; padding: 2px; display: inline-block; margin-right: 5px;">Deprecated</div> <div data-bbox="137 640 786 712" style="border: 1px solid #ccc; border-radius: 3px; padding: 5px; margin-top: 10px;"> <b>USING</b> BTREE INDEX </div>	<p>B-tree indexes are deprecated.</p> <p>Replaced by:</p> <div data-bbox="807 719 1457 790" style="border: 1px solid #ccc; border-radius: 3px; padding: 5px; margin-top: 10px;"> <b>USING</b> {POINT   RANGE   TEXT} INDEX </div>
<div data-bbox="137 837 477 887" style="border: 1px solid #0070C0; border-radius: 3px; padding: 2px; display: inline-block; margin-right: 5px;">Functionality</div> <div data-bbox="320 837 477 887" style="border: 1px solid #0070C0; border-radius: 3px; padding: 2px; display: inline-block; margin-right: 5px;">Deprecated</div> <div data-bbox="137 916 786 1039" style="border: 1px solid #ccc; border-radius: 3px; padding: 5px; margin-top: 10px;"> <b>CREATE CONSTRAINT</b>  ...  OPTIONS {" btree-option: btree-value[, ...] "} </div>	<p>Node key and uniqueness constraints with b-tree options are deprecated and will be replaced in 5.0 by range options, see <a href="#">range indexes</a>. In 4.4, the b-tree index-backed constraints are still the correct alternative to use.</p> <p>Will be replaced by:</p> <div data-bbox="807 1160 1457 1283" style="border: 1px solid #ccc; border-radius: 3px; padding: 5px; margin-top: 10px;"> <b>CREATE CONSTRAINT</b>  ...  OPTIONS {" range-option: range-value[, ...] "} </div> <p>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 <a href="#">point indexes</a>.</p>
<div data-bbox="137 1621 477 1671" style="border: 1px solid #0070C0; border-radius: 3px; padding: 2px; display: inline-block; margin-right: 5px;">Functionality</div> <div data-bbox="320 1621 477 1671" style="border: 1px solid #0070C0; border-radius: 3px; padding: 2px; display: inline-block; margin-right: 5px;">Deprecated</div> <div data-bbox="137 1700 786 1771" style="border: 1px solid #ccc; border-radius: 3px; padding: 5px; margin-top: 10px;"> distance(n.prop, point({x:0, y:0})) </div>	<p>Replaced by:</p> <div data-bbox="807 1700 1457 1771" style="border: 1px solid #ccc; border-radius: 3px; padding: 5px; margin-top: 10px;"> point.distance(n.prop, point({x:0, y:0})) </div>

Feature	Details
<p data-bbox="140 163 475 208">Functionality <span>Deprecated</span></p> <pre data-bbox="140 241 786 331">point({x:0, y:0}) &lt;= point({x:1, y:1}) &lt;= point({x:2, y:2})</pre>	<p data-bbox="802 163 1385 241">The ability to use the &lt;, &lt;=, &gt;, and &gt;= on spatial points is deprecated.</p> <p data-bbox="802 286 954 320">Instead use:</p> <pre data-bbox="802 376 1457 454">point.withinBBox(point({x:1, y:1}), point({x:0, y:0}), point({x:2, y:2}))</pre>
<p data-bbox="140 510 443 555">Procedure <span>Deprecated</span></p> <pre data-bbox="140 589 786 656">dbms.listTransactions</pre>	<p data-bbox="802 510 970 544">Replaced by:</p> <pre data-bbox="802 600 1457 768">SHOW TRANSACTION[S] [transaction-id[,...]] [YIELD { *   field[, ...] } [ORDER BY field[, ...]] [SKIP n] [LIMIT n]] [WHERE expression] [RETURN field[, ...] [ORDER BY field[, ...]] [SKIP n] [LIMIT n]]</pre>
<p data-bbox="140 842 443 887">Procedure <span>Deprecated</span></p> <pre data-bbox="140 920 786 987">dbms.killTransaction</pre> <pre data-bbox="140 1021 786 1088">dbms.killTransactions</pre>	<p data-bbox="802 842 970 875">Replaced by:</p> <pre data-bbox="802 931 1457 987">TERMINATE TRANSACTION[S] transaction-id[,...]</pre>
<p data-bbox="140 1149 443 1193">Procedure <span>Deprecated</span></p> <pre data-bbox="140 1227 786 1294">dbms.listQueries</pre>	<p data-bbox="802 1149 970 1182">Replaced by:</p> <pre data-bbox="802 1238 1457 1406">SHOW TRANSACTION[S] [transaction-id[,...]] [YIELD { *   field[, ...] } [ORDER BY field[, ...]] [SKIP n] [LIMIT n]] [WHERE expression] [RETURN field[, ...] [ORDER BY field[, ...]] [SKIP n] [LIMIT n]]</pre>
<p data-bbox="140 1480 443 1525">Procedure <span>Deprecated</span></p> <pre data-bbox="140 1559 786 1626">dbms.killQuery</pre> <pre data-bbox="140 1659 786 1727">dbms.killQueries</pre>	<p data-bbox="802 1480 970 1514">Replaced by:</p> <pre data-bbox="802 1559 1457 1615">TERMINATE TRANSACTION[S] transaction-id[,...]</pre>

## New features

Feature	Details
<p><b>Functionality</b> <b>New</b></p> <pre>CALL {   ... } IN TRANSACTIONS</pre>	<p>New clause for evaluating a subquery in separate transactions. Typically used when modifying or importing large amounts of data. See <code>CALL { ... } IN TRANSACTIONS</code>.</p>
<p><b>Syntax</b> <b>New</b></p> <pre>CREATE CONSTRAINT FOR ... REQUIRE ...</pre>	<p>New syntax for creating constraints, applicable to all constraint types.</p>
<p><b>Functionality</b> <b>New</b></p> <pre>CREATE CONSTRAINT [constraint_name] [IF NOT EXISTS] FOR (n:LabelName) REQUIRE (n.propertyName_1, ..., n.propertyName_n) IS UNIQUE [OPTIONS {"option": value[, ...]}]</pre>	<p>Unique property constraints now allow multiple properties, ensuring that the combination of property values are unique.</p>
<p><b>Functionality</b> <b>New</b> <b>Deprecated</b></p> <pre>DROP CONSTRAINT ON (n:LabelName) ASSERT (n.propertyName_1, ..., n.propertyName_n) IS UNIQUE</pre>	<p>Unique property constraints now allow multiple properties.</p> <p>Replaced by:</p> <pre>DROP CONSTRAINT name [IF EXISTS]</pre>
<p><b>Syntax</b> <b>New</b></p> <pre>CREATE CONSTRAINT [constraint_name] [IF NOT EXISTS] FOR ... REQUIRE ... IS NOT NULL OPTIONS {" "}"</pre>	<p>Existence constraints now allow an <code>OPTIONS</code> map, however, at this point there are no available values for the map.</p>
<p><b>Functionality</b> <b>New</b></p> <pre>CREATE LOOKUP INDEX [index_name] [IF NOT EXISTS] FOR ... ON ... OPTIONS {"option": value[, ...]}]</pre>	<p>Token lookup indexes now allow an <code>OPTIONS</code> map to specify the index provider.</p>
<p><b>Functionality</b> <b>New</b></p> <pre>CREATE TEXT INDEX ...</pre>	<p>Allows creating text indexes on nodes or relationships with a particular label or relationship type, and property combination. They can be dropped by using their name.</p>

Feature	Details
<p>Functionality <span>New</span></p> <pre>CREATE RANGE INDEX ...</pre>	<p><b>FUTURE INDEX:</b> Allows creating range indexes on nodes or relationships with a particular label or relationship type, and properties combination. They can be dropped by using their name.</p>
<p>Functionality <span>New</span></p> <pre>CREATE CONSTRAINT ... OPTIONS {" indexProvider: 'range-1.0' }"</pre>	<p><b>FUTURE CONSTRAINT:</b> Allows creating node key and uniqueness constraints backed by range indexes by providing the range index provider in the <b>OPTIONS</b> map.</p>
<p>Functionality <span>New</span></p> <pre>CREATE POINT INDEX ...</pre>	<p><b>FUTURE INDEX:</b> Allows creating point indexes on nodes or relationships with a particular label or relationship type, and property combination. They can be dropped by using their name.</p>
<p>Syntax <span>New</span> New privilege:</p> <pre>IMPERSONATE</pre>	<p>New privilege that allows a user to assume privileges of another one.</p>
<p>Functionality <span>New</span></p> <pre>SHOW TRANSACTION[S] [transaction-id[,...]] [YIELD { *   field[, ...] } [ORDER BY field[, ...]] [SKIP n] [LIMIT n]] [WHERE expression] [RETURN field[, ...] [ORDER BY field[, ...]] [SKIP n] [LIMIT n]]</pre>	<p>List transactions on the current server.</p> <p>The <b>transaction-id</b> is a comma-separated list of one or more quoted strings, a string parameter, or a list parameter.</p>
<p>Functionality <span>New</span></p> <pre>TERMINATE TRANSACTION[S] transaction-id[,...]</pre>	<p>Terminate transactions on the current server.</p> <p>The <b>transaction-id</b> is a comma-separated list of one or more quoted strings, a string parameter, or a list parameter.</p>
<p>Functionality <span>New</span></p> <pre>ALTER DATABASE ... [IF EXISTS] SET ACCESS {READ ONLY   READ WRITE}</pre>	<p>New Cypher command for modifying a database by changing its access mode.</p>

Feature	Details
<p><b>Functionality</b> <b>New</b> New privilege:</p> <pre>ALTER DATABASE</pre>	<p>New privilege that allows a user to modify databases.</p>
<p><b>Functionality</b> <b>New</b> New privilege:</p> <pre>SET DATABASE ACCESS</pre>	<p>New privilege that allows a user to modify database access mode.</p>
<p><b>Functionality</b> <b>New</b></p> <pre>CREATE ALIAS ... [IF NOT EXISTS] FOR DATABASE ...</pre>	<p>New Cypher command for creating an alias for a database name. Remote aliases are only supported from version 4.4.8.</p>
<p><b>Functionality</b> <b>New</b></p> <pre>CREATE OR REPLACE ALIAS ... FOR DATABASE ...</pre>	<p>New Cypher command for creating or replacing an alias for a database name. Remote aliases are only supported from version 4.4.8.</p>
<p><b>Functionality</b> <b>New</b></p> <pre>ALTER ALIAS ... [IF EXISTS] SET DATABASE ...</pre>	<p>New Cypher command for altering an alias. Remote aliases are only supported from version 4.4.8.</p>
<p><b>Functionality</b> <b>New</b></p> <pre>DROP ALIAS ... [IF EXISTS] FOR DATABASE</pre>	<p>New Cypher command for dropping a database alias.</p>
<p><b>Functionality</b> <b>New</b></p> <pre>SHOW ALIASES FOR DATABASE</pre>	<p>New Cypher command for listing database aliases. Only supported since version 4.4.8.</p>
<p><b>Functionality</b> <b>New</b> New privilege:</p> <pre>ALIAS MANAGEMENT</pre>	<p>New privilege that allows a user to create, modify, delete and list aliases. Only supported since version 4.4.8.</p>
<p><b>Functionality</b> <b>New</b> New privilege:</p> <pre>CREATE ALIAS</pre>	<p>New privilege that allows a user to create aliases. Only supported since version 4.4.8.</p>

Feature	Details
<p><b>Functionality</b> <b>New</b> New privilege:</p> <pre>ALTER ALIAS</pre>	New privilege that allows a user to modify aliases. Only supported since version 4.4.8.
<p><b>Functionality</b> <b>New</b> New privilege:</p> <pre>DROP ALIAS</pre>	New privilege that allows a user to delete aliases. Only supported since version 4.4.8.
<p><b>Functionality</b> <b>New</b> New privilege:</p> <pre>SHOW ALIAS</pre>	New privilege that allows a user to show aliases. Only supported since version 4.4.8.

## Version 4.3

### Deprecated features

Feature	Details
<p><b>Syntax</b> <b>Deprecated</b></p> <pre>CREATE CONSTRAINT [name] ON (node:Label) ASSERT exists(node.property)</pre>	<p>Replaced by:</p> <pre>CREATE CONSTRAINT [name] ON (node:Label) ASSERT node.property IS NOT NULL</pre>
<p><b>Syntax</b> <b>Deprecated</b></p> <pre>CREATE CONSTRAINT [name] ON ()-[rel:REL]-() ASSERT exists(rel.property)</pre>	<p>Replaced by:</p> <pre>CREATE CONSTRAINT [name] ON ()-[rel:REL]-() ASSERT rel.property IS NOT NULL</pre>
<p><b>Syntax</b> <b>Deprecated</b></p> <pre>exists(prop)</pre>	<p>Replaced by:</p> <pre>prop IS NOT NULL</pre>
<p><b>Syntax</b> <b>Deprecated</b></p> <pre>NOT exists(prop)</pre>	<p>Replaced by:</p> <pre>prop IS NULL</pre>

Feature	Details
<p><b>Syntax</b> <b>Deprecated</b> BRIEF [OUTPUT] for SHOW INDEXES and SHOW CONSTRAINTS.</p>	<p>Replaced by default output columns.</p>
<p><b>Syntax</b> <b>Deprecated</b> VERBOSE [OUTPUT] for SHOW INDEXES and SHOW CONSTRAINTS.</p>	<p>Replaced by:</p> <pre>YIELD *</pre>
<p><b>Syntax</b> <b>Deprecated</b></p> <pre>SHOW EXISTS CONSTRAINTS</pre>	<p>Replaced by:</p> <pre>SHOW [PROPERTY] EXIST[ENCE] CONSTRAINTS</pre> <p>Still allows BRIEF and VERBOSE but not YIELD or WHERE.</p>
<p><b>Syntax</b> <b>Deprecated</b></p> <pre>SHOW NODE EXISTS CONSTRAINTS</pre>	<p>Replaced by:</p> <pre>SHOW NODE [PROPERTY] EXIST[ENCE] CONSTRAINTS</pre> <p>Still allows BRIEF and VERBOSE but not YIELD or WHERE.</p>
<p><b>Syntax</b> <b>Deprecated</b></p> <pre>SHOW RELATIONSHIP EXISTS CONSTRAINTS</pre>	<p>Replaced by:</p> <pre>SHOW RELATIONSHIP [PROPERTY] EXIST[ENCE] CONSTRAINTS</pre> <p>Still allows BRIEF and VERBOSE but not YIELD or WHERE.</p>
<p><b>Syntax</b> <b>Deprecated</b> For privilege commands:</p> <pre>ON DEFAULT DATABASE</pre>	<p>Replaced by:</p> <pre>ON HOME DATABASE</pre>
<p><b>Syntax</b> <b>Deprecated</b> For privilege commands:</p> <pre>ON DEFAULT GRAPH</pre>	<p>Replaced by:</p> <pre>ON HOME GRAPH</pre>



Feature	Details
<p><b>Syntax</b> <b>Deprecated</b></p> <pre>MATCH (a) RETURN (a)--()</pre>	<p>Pattern expressions producing lists of paths are deprecated, but they can still be used as existence predicates, for example in <b>WHERE</b> clauses.</p> <p>Instead, use a pattern comprehension:</p> <pre>MATCH (a) RETURN [p=(a)--()   p]</pre>
<p><b>Procedure</b> <b>Deprecated</b></p> <pre>dbms.procedures</pre>	<p>Replaced by:</p> <pre>SHOW PROCEDURE[S] [EXECUTABLE [BY {CURRENT USER   username}]] [YIELD ...] [WHERE ...] [RETURN ...]</pre>
<p><b>Procedure</b> <b>Deprecated</b></p> <pre>dbms.functions</pre>	<p>Replaced by:</p> <pre>SHOW [ALL   BUILT IN   USER DEFINED] FUNCTION[S] [EXECUTABLE [BY {CURRENT USER   username}]] [YIELD ...] [WHERE ...] [RETURN ...]</pre>

## Updated features

Feature	Details
<p><b>Functionality</b> <b>Updated</b></p> <pre>SHOW INDEXES WHERE ...</pre>	<p>Now allows filtering for:</p> <pre>SHOW INDEXES</pre>
<p><b>Functionality</b> <b>Updated</b></p> <pre>SHOW CONSTRAINTS WHERE ...</pre>	<p>Now allows filtering for:</p> <pre>SHOW CONSTRAINTS</pre>
<p><b>Functionality</b> <b>Updated</b></p> <pre>SHOW INDEXES YIELD ... [WHERE ...] [RETURN ...]</pre>	<p>Now allows <b>YIELD</b>, <b>WHERE</b>, and <b>RETURN</b> clauses to <b>SHOW INDEXES</b> to change the output.</p>

Feature	Details
<p><b>Functionality</b> <b>Updated</b></p> <pre>SHOW CONSTRAINTS <b>YIELD</b> ... <b>[WHERE ...]</b> <b>[RETURN ...]</b></pre>	<p>Now allows <b>YIELD</b>, <b>WHERE</b>, and <b>RETURN</b> clauses to <b>SHOW CONSTRAINTS</b> to change the output.</p>
<p><b>Syntax</b> <b>Updated</b></p> <pre>SHOW <b>[PROPERTY]</b> <b>EXIST[ENCE]</b> CONSTRAINTS</pre>	<p>New syntax for filtering <b>SHOW CONSTRAINTS</b> on property existence constraints. Allows <b>YIELD</b> and <b>WHERE</b> but not <b>BRIEF</b> or <b>VERBOSE</b>.</p>
<p><b>Syntax</b> <b>Updated</b></p> <pre>SHOW <b>NODE</b> <b>[PROPERTY]</b> <b>EXIST[ENCE]</b> CONSTRAINTS</pre>	<p>New syntax for filtering <b>SHOW CONSTRAINTS</b> on node property existence constraints. Allows <b>YIELD</b> and <b>WHERE</b> but not <b>BRIEF</b> or <b>VERBOSE</b>.</p>
<p><b>Syntax</b> <b>Updated</b></p> <pre>SHOW <b>REL[ATIONSHIP]</b> <b>[PROPERTY]</b> <b>EXIST[ENCE]</b> CONSTRAINTS</pre>	<p>New syntax for filtering <b>SHOW CONSTRAINTS</b> on relationship property existence constraints. Allows <b>YIELD</b> and <b>WHERE</b> but not <b>BRIEF</b> or <b>VERBOSE</b>.</p>
<p><b>Functionality</b> <b>Updated</b></p> <pre>SHOW FULLTEXT INDEXES</pre>	<p>Now allows easy filtering for <b>SHOW INDEXES</b> on fulltext indexes. Allows <b>YIELD</b> and <b>WHERE</b> but not <b>BRIEF</b> or <b>VERBOSE</b>.</p>
<p><b>Functionality</b> <b>Updated</b></p> <pre>SHOW LOOKUP INDEXES</pre>	<p>Now allows easy filtering for <b>SHOW INDEXES</b> on token lookup indexes. Allows <b>YIELD</b> and <b>WHERE</b> but not <b>BRIEF</b> or <b>VERBOSE</b>.</p>

## New features

Feature	Details
<p><b>Syntax</b> <b>New</b></p> <pre><b>CREATE</b> DATABASE ... <b>[OPTIONS {...}]</b></pre>	<p>New syntax to pass options to <b>CREATE DATABASE</b>. This can be used to specify a specific cluster node to seed data from.</p>

Feature	Details
<p><a href="#">Syntax</a> <a href="#">New</a></p> <pre>CREATE CONSTRAINT [name] ON (node:Label) ASSERT node.property IS NOT NULL</pre>	<p>New syntax for creating node property existence constraints.</p>
<p><a href="#">Syntax</a> <a href="#">New</a></p> <pre>CREATE CONSTRAINT [name] ON ()-[rel:REL]-() ASSERT rel.property IS NOT NULL</pre>	<p>New syntax for creating relationship property existence constraints.</p>
<p><a href="#">Syntax</a> <a href="#">New</a></p> <pre>ALTER USER name IF EXISTS ...</pre>	<p>Makes altering users idempotent. If the specified name does not exist, no error is thrown.</p>
<p><a href="#">Syntax</a> <a href="#">New</a></p> <pre>ALTER USER ... SET HOME DATABASE ...</pre>	<p>Now allows setting home database for user.</p>
<p><a href="#">Syntax</a> <a href="#">New</a></p> <pre>ALTER USER ... REMOVE HOME DATABASE</pre>	<p>Now allows removing home database for user.</p>
<p><a href="#">Syntax</a> <a href="#">New</a></p> <pre>CREATE USER ... SET HOME DATABASE ...</pre>	<p><b>CREATE USER</b> now allows setting home database for user.</p>
<p><a href="#">Syntax</a> <a href="#">New</a></p> <pre>SHOW HOME DATABASE</pre>	<p>New syntax for showing the home database of the current user.</p>
<p><a href="#">Syntax</a> <a href="#">New</a> New privilege:</p> <pre>SET USER HOME DATABASE</pre>	<p>New Cypher command for administering privilege for changing users home database.</p>

Feature	Details
<p><b>Syntax</b> <b>New</b> For privilege commands:</p> <pre>ON HOME DATABASE</pre>	<p>New syntax for privileges affecting home database.</p>
<p><b>Syntax</b> <b>New</b> For privilege commands:</p> <pre>ON HOME GRAPH</pre>	<p>New syntax for privileges affecting home graph.</p>
<p><b>Syntax</b> <b>New</b></p> <pre>CREATE FULLTEXT INDEX ...</pre>	<p>Allows creating fulltext indexes on nodes or relationships. They can be dropped by using their name.</p>
<p><b>Functionality</b> <b>New</b></p> <pre>CREATE INDEX FOR ()-[r:TYPE]-() ...</pre>	<p>Allows creating indexes on relationships with a particular relationship type and property combination. They can be dropped by using their name.</p>
<p><b>Functionality</b> <b>New</b></p> <pre>CREATE LOOKUP INDEX ...</pre>	<p>Create token lookup index for nodes with any labels or relationships with any relationship type. They can be dropped by using their name.</p>
<p><b>Functionality</b> <b>New</b></p> <pre>RENAME ROLE</pre>	<p>New Cypher command for changing the name of a role.</p>
<p><b>Functionality</b> <b>New</b></p> <pre>RENAME USER</pre>	<p>New Cypher command for changing the name of a user.</p>
<p><b>Functionality</b> <b>New</b></p> <pre>SHOW PROCEDURE[S] [EXECUTABLE [BY {CURRENT USER   username}]] [YIELD ...] [WHERE ...] [RETURN ...]</pre>	<p>New Cypher commands for listing procedures.</p>

Feature	Details
<p><b>Functionality</b> <b>New</b></p> <pre>SHOW [ALL   BUILT IN   USER DEFINED] FUNCTION[S] [EXECUTABLE [BY {CURRENT USER   username}]] [YIELD ...] [WHERE ...] [RETURN ...]</pre>	New Cypher commands for listing functions.

## Version 4.2

### Deprecated features

Feature	Details
<p><b>Syntax</b> <b>Deprecated</b></p> <pre>0...</pre>	Replaced by <code>0o...</code>
<p><b>Syntax</b> <b>Deprecated</b></p> <pre>0X...</pre>	Only <code>0x...</code> (lowercase x) is supported.
<p><b>Procedure</b> <b>Deprecated</b></p> <pre>db.createIndex</pre>	Replaced by <code>CREATE INDEX</code> command.
<p><b>Procedure</b> <b>Deprecated</b></p> <pre>db.createNodeKey</pre>	Replaced by: <code>CREATE CONSTRAINT ... IS NODE KEY</code>
<p><b>Procedure</b> <b>Deprecated</b></p> <pre>db.createUniquePropertyConstraint</pre>	Replaced by: <code>CREATE CONSTRAINT ... IS UNIQUE</code>
<p><b>Procedure</b> <b>Deprecated</b></p> <pre>db.indexes</pre>	Replaced by: <code>SHOW INDEXES</code>

Feature	Details
<p><b>Procedure</b> <b>Deprecated</b></p> <p>db.indexDetails</p>	<p>Replaced by:</p> <p>SHOW INDEXES <b>YIELD</b> *</p>
<p><b>Procedure</b> <b>Deprecated</b></p> <p>db.constraints</p>	<p>Replaced by:</p> <p>SHOW CONSTRAINTS</p>
<p><b>Procedure</b> <b>Deprecated</b></p> <p>db.schemaStatements</p>	<p>Replaced by:</p> <p>SHOW INDEXES <b>YIELD</b> *</p> <p>SHOW CONSTRAINTS <b>YIELD</b> *</p>
<p><b>Syntax</b> <b>Deprecated</b></p> <p>CALL { RETURN 1 }</p>	<p>Replaced by:</p> <p>CALL { RETURN 1 AS one }</p> <p>Unaliased expressions are deprecated in subquery <b>RETURN</b> clauses.</p>

## Updated features

Feature	Details
<p><b>Functionality</b> <b>Updated</b></p> <p>SHOW ROLE name PRIVILEGES</p>	<p>Can now handle multiple roles.</p> <p>SHOW ROLES n1, n2, ... PRIVILEGES</p>
<p><b>Functionality</b> <b>Updated</b></p> <p>SHOW USER name PRIVILEGES</p>	<p>Can now handle multiple users.</p> <p>SHOW USERS n1, n2, ... PRIVILEGES</p>
<p><b>Functionality</b> <b>Updated</b></p> <p>round(expression, precision)</p>	<p>The <b>round()</b> function can now take an additional argument to specify rounding precision.</p>

Feature	Details
<p><b>Functionality</b> <b>Updated</b></p> <pre>round(expression, precision, mode)</pre>	<p>The <code>round()</code> function can now take two additional arguments to specify rounding precision and rounding mode.</p>

## New features

Feature	Details
<p><b>Functionality</b> <b>New</b></p> <pre>SHOW PRIVILEGES [AS [REVOKE] COMMAND[S]]</pre>	<p>Privileges can now be shown as Cypher commands.</p>
<p><b>Syntax</b> <b>New</b></p> <pre>DEFAULT GRAPH</pre>	<p>New optional part of the Cypher commands for <a href="#">database privileges</a>.</p>
<p><b>Syntax</b> <b>New</b></p> <pre>0o...</pre>	<p>Cypher now interprets literals with prefix <code>0o</code> as an octal integer literal.</p>
<p><b>Syntax</b> <b>New</b></p> <pre>SET [PLAINTEXT   ENCRYPTED] PASSWORD</pre>	<p>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.</p>
<p><b>Functionality</b> <b>New</b> New privilege:</p> <pre>EXECUTE</pre>	<p>New Cypher commands for administering privileges for executing procedures and user defined functions. See <a href="#">The DBMS EXECUTE privileges</a>.</p>
<p><b>Syntax</b> <b>New</b></p> <pre>CREATE [BTREE] INDEX ... [OPTIONS {...}]</pre>	<p>Allows setting index provider and index configuration when creating an index.</p>
<p><b>Syntax</b> <b>New</b></p> <pre>CREATE CONSTRAINT ... IS NODE KEY [OPTIONS {...}]</pre>	<p>Allows setting index provider and index configuration for the backing index when creating a node key constraint.</p>

Feature	Details
<p>Syntax <b>New</b></p> <pre>CREATE CONSTRAINT ... IS UNIQUE [OPTIONS {...}]</pre>	Allows setting index provider and index configuration for the backing index when creating a uniqueness constraint.
<p>Syntax <b>New</b></p> <pre>SHOW CURRENT USER</pre>	New Cypher command for showing current logged-in user and roles.
<p>Functionality <b>New</b></p> <pre>SHOW [ALL   BTREE] INDEX[ES] [BRIEF   VERBOSE [OUTPUT]]</pre>	New Cypher commands for listing indexes.
<p>Functionality <b>New</b></p> <pre>SHOW [ALL   UNIQUE   NODE EXIST[S]   RELATIONSHIP EXIST[S]   EXIST[S]   NODE KEY] CONSTRAINT[S] [BRIEF   VERBOSE [OUTPUT]]</pre>	New Cypher commands for listing constraints.
<p>Functionality <b>New</b> New privilege:</p> <pre>SHOW INDEX</pre>	New Cypher command for administering privilege for listing indexes.
<p>Functionality <b>New</b> New privilege:</p> <pre>SHOW CONSTRAINT</pre>	New Cypher command for administering privilege for listing constraints.

## Version 4.1.3

### New features

Feature	Details
<p>Syntax <b>New</b></p> <pre>CREATE INDEX [name] IF NOT EXISTS FOR ...</pre>	Makes index creation idempotent. If an index with the name or schema already exists no error will be thrown.



Feature	Details
<p>Syntax New</p> <pre>DROP INDEX name IF EXISTS</pre>	Makes index deletion idempotent. If no index with the name exists no error will be thrown.
<p>Syntax New</p> <pre>CREATE CONSTRAINT [name] IF NOT EXISTS ON ...</pre>	Makes constraint creation idempotent. If a constraint with the name or type and schema already exists no error will be thrown.
<p>Syntax New</p> <pre>DROP CONSTRAINT name IF EXISTS</pre>	Makes constraint deletion idempotent. If no constraint with the name exists no error will be thrown.

## Version 4.1

### Restricted features

Feature	Details
<p>Functionality Restricted</p> <pre>REVOKE ...</pre>	No longer revokes sub-privileges when revoking a compound privilege, e.g. when revoking <b>INDEX MANAGEMENT</b> , any <b>CREATE INDEX</b> and <b>DROP INDEX</b> privileges will no longer be revoked.
<p>Functionality Restricted</p> <pre>ALL DATABASE PRIVILEGES</pre>	No longer includes the privileges <b>START DATABASE</b> and <b>STOP DATABASE</b> .

### Updated features

Feature	Details
<p>Procedure Updated</p> <pre>queryId</pre>	<p>The <b>queryId</b> procedure format has changed, and no longer includes the database name. For example, <b>mydb-query-123</b> is now <b>query-123</b>.</p> <p>This change affects the procedures:  <b>dbms.listQueries()</b>,  <b>dbms.listActiveLocks(queryId)</b>,  <b>dbms.killQueries(queryIds)</b>, and  <b>dbms.killQuery(queryId)</b>.</p>

Feature	Details
<p><b>Functionality</b> <b>Updated</b></p> <p>SHOW PRIVILEGES</p>	<p>The returned privileges are a closer match to the original grants and denies, e.g. if granted <b>MATCH</b> the command will show that specific privilege and not the <b>TRAVERSE</b> and <b>READ</b> privileges. Added support for <b>YIELD</b> and <b>WHERE</b> clauses to allow filtering results.</p>

## New features

Feature	Details
<p><b>Functionality</b> <b>New</b> New role:</p> <p>PUBLIC</p>	<p>The <b>PUBLIC</b> role is automatically assigned to all users, giving them a set of base privileges.</p>
<p><b>Syntax</b> <b>New</b> For privileges:</p> <p>REVOKE <b>MATCH</b></p>	<p>The <b>MATCH</b> privilege can now be revoked.</p>
<p><b>Functionality</b> <b>New</b></p> <p>SHOW USERS</p>	<p>New support for <b>YIELD</b> and <b>WHERE</b> clauses to allow filtering results.</p>
<p><b>Functionality</b> <b>New</b></p> <p>SHOW ROLES</p>	<p>New support for <b>YIELD</b> and <b>WHERE</b> clauses to allow filtering results.</p>
<p><b>Functionality</b> <b>New</b></p> <p>SHOW DATABASES</p>	<p>New support for <b>YIELD</b> and <b>WHERE</b> clauses to allow filtering results.</p>
<p><b>Functionality</b> <b>New</b> TRANSACTION MANAGEMENT privileges</p>	<p>New Cypher commands for administering transaction management.</p>
<p><b>Functionality</b> <b>New</b> DBMS USER MANAGEMENT privileges</p>	<p>New Cypher commands for administering user management.</p>
<p><b>Functionality</b> <b>New</b> DBMS DATABASE MANAGEMENT privileges</p>	<p>New Cypher commands for administering database management.</p>

Feature	Details
<p><b>Functionality</b> <b>New</b></p> <p>DBMS <code>PRIVILEGE MANAGEMENT</code> privileges</p>	New Cypher commands for administering privilege management.
<p><b>Functionality</b> <b>New</b></p> <p><code>ALL DBMS PRIVILEGES</code></p>	New Cypher command for administering role, user, database and privilege management.
<p><b>Functionality</b> <b>New</b></p> <p><code>ALL GRAPH PRIVILEGES</code></p>	New Cypher command for administering read and write privileges.
<p><b>Functionality</b> <b>New</b> Write privileges</p>	New Cypher commands for administering write privileges.
<p><b>Functionality</b> <b>New</b></p> <p><code>ON DEFAULT DATABASE</code></p>	New optional part of the Cypher commands for <a href="#">database privileges</a> .

## Version 4.0

### Removed features

Feature	Details
<p><b>Function</b> <b>Removed</b></p> <p><code>rels()</code></p>	<p>Replaced by:</p> <p><code>relationships()</code></p> <p>See <code>relationships()</code>.</p>
<p><b>Function</b> <b>Removed</b></p> <p><code>toInt()</code></p>	<p>Replaced by:</p> <p><code>toInteger()</code></p> <p>See <code>toInteger()</code>.</p>

Feature	Details
<div data-bbox="137 159 397 210" style="border: 1px solid #ccc; padding: 2px; display: inline-block; margin-bottom: 5px;">Function</div> <div data-bbox="264 159 397 210" style="border: 1px solid #ccc; padding: 2px; display: inline-block; margin-left: 5px; background-color: #e0f0ff;">Removed</div> <div data-bbox="137 237 786 311" style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <code>lower()</code> </div>	<p>Replaced by:</p> <div data-bbox="802 237 1458 311" style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <code>toLowerCase()</code> </div> <p>See <code>toLowerCase()</code>.</p>
<div data-bbox="137 439 397 490" style="border: 1px solid #ccc; padding: 2px; display: inline-block; margin-bottom: 5px;">Function</div> <div data-bbox="264 439 397 490" style="border: 1px solid #ccc; padding: 2px; display: inline-block; margin-left: 5px; background-color: #e0f0ff;">Removed</div> <div data-bbox="137 517 786 591" style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <code>upper()</code> </div>	<p>Replaced by:</p> <div data-bbox="802 517 1458 591" style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <code>toUpperCase()</code> </div> <p>See <code>toUpperCase()</code>.</p>
<div data-bbox="137 719 397 770" style="border: 1px solid #ccc; padding: 2px; display: inline-block; margin-bottom: 5px;">Function</div> <div data-bbox="264 719 397 770" style="border: 1px solid #ccc; padding: 2px; display: inline-block; margin-left: 5px; background-color: #e0f0ff;">Removed</div> <div data-bbox="137 797 786 871" style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <code>extract()</code> </div>	<p>Replaced by <a href="#">list comprehension</a>.</p>
<div data-bbox="137 916 397 967" style="border: 1px solid #ccc; padding: 2px; display: inline-block; margin-bottom: 5px;">Function</div> <div data-bbox="264 916 397 967" style="border: 1px solid #ccc; padding: 2px; display: inline-block; margin-left: 5px; background-color: #e0f0ff;">Removed</div> <div data-bbox="137 999 786 1072" style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <code>filter()</code> </div>	<p>Replaced by <a href="#">list comprehension</a>.</p>
<div data-bbox="137 1113 448 1164" style="border: 1px solid #ccc; padding: 2px; display: inline-block; margin-bottom: 5px;">Functionality</div> <div data-bbox="320 1113 448 1164" style="border: 1px solid #ccc; padding: 2px; display: inline-block; margin-left: 5px; background-color: #e0f0ff;">Removed</div> <div data-bbox="456 1126 671 1160" style="margin-left: 5px;">For Rule planner:</div> <div data-bbox="137 1196 786 1270" style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <code>CYPHER planner=rule</code> </div>	<p>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.</p>
<div data-bbox="137 1310 448 1361" style="border: 1px solid #ccc; padding: 2px; display: inline-block; margin-bottom: 5px;">Functionality</div> <div data-bbox="320 1310 448 1361" style="border: 1px solid #ccc; padding: 2px; display: inline-block; margin-left: 5px; background-color: #e0f0ff;">Removed</div> <div data-bbox="456 1323 651 1357" style="margin-left: 5px;">Explicit indexes</div>	<p>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 <a href="#">built-in procedures for Neo4j 3.3 to 3.5</a>.</p>
<div data-bbox="137 1547 448 1599" style="border: 1px solid #ccc; padding: 2px; display: inline-block; margin-bottom: 5px;">Functionality</div> <div data-bbox="320 1547 448 1599" style="border: 1px solid #ccc; padding: 2px; display: inline-block; margin-left: 5px; background-color: #e0f0ff;">Removed</div> <div data-bbox="456 1561 730 1594" style="margin-left: 5px;">For compiled runtime:</div> <div data-bbox="137 1626 786 1700" style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <code>CYPHER runtime=compiled</code> </div>	<p>Replaced by the new <code>pipelined</code> runtime which covers a much wider range of queries.</p>
<div data-bbox="137 1744 371 1796" style="border: 1px solid #ccc; padding: 2px; display: inline-block; margin-bottom: 5px;">Clause</div> <div data-bbox="240 1744 371 1796" style="border: 1px solid #ccc; padding: 2px; display: inline-block; margin-left: 5px; background-color: #e0f0ff;">Removed</div> <div data-bbox="137 1827 786 1901" style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <code>CREATE UNIQUE</code> </div>	<p>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.</p>

Feature	Details
<p>Clause <b>Removed</b></p> <pre>START</pre>	<p>Running queries with this clause will cause a syntax error. Running with <b>CYPHER 3.5</b> will cause a runtime error due to the removal of the rule planner.</p>
<p>Syntax <b>Removed</b></p> <pre>MATCH (n)-[:A B C {foo: 'bar'}]-() RETURN n</pre>	<p>Replaced by:</p> <pre>MATCH (n)-[:A B C {foo: 'bar'}]-() RETURN n</pre>
<p>Syntax <b>Removed</b></p> <pre>MATCH (n)-[x:A B C]-() RETURN n</pre>	<p>Replaced by:</p> <pre>MATCH (n)-[x:A B C]-() RETURN n</pre>
<p>Syntax <b>Removed</b></p> <pre>MATCH (n)-[x:A B C*]-() RETURN n</pre>	<p>Replaced by:</p> <pre>MATCH (n)-[x:A B C*]-() RETURN n</pre>
<p>Syntax <b>Removed</b></p> <pre>{parameter}</pre>	<p>Replaced by:</p> <pre>\$parameter</pre> <p>See <a href="#">Parameters</a>.</p>

## Deprecated features

Feature	Details
<p>Syntax <b>Deprecated</b></p> <pre>MATCH (n)-[rs*]-() RETURN rs</pre>	<p>As in Cypher 3.2, this is replaced by:</p> <pre>MATCH p=(n)-[*]-() RETURN relationships(p) AS rs</pre>
<p>Syntax <b>Deprecated</b></p> <pre>CREATE INDEX ON :Label(prop)</pre>	<p>Replaced by:</p> <pre>CREATE INDEX FOR (n:Label) ON (n.prop)</pre>
<p>Syntax <b>Deprecated</b></p> <pre>DROP INDEX ON :Label(prop)</pre>	<p>Replaced by:</p> <pre>DROP INDEX name</pre>

Feature	Details
<p><b>Syntax</b> <b>Deprecated</b></p> <pre>DROP CONSTRAINT ON (n:Label) ASSERT (n.prop) IS NODE KEY</pre>	<p>Replaced by:</p> <pre>DROP CONSTRAINT name</pre>
<p><b>Syntax</b> <b>Deprecated</b></p> <pre>DROP CONSTRAINT ON (n:Label) ASSERT (n.prop) IS UNIQUE</pre>	<p>Replaced by:</p> <pre>DROP CONSTRAINT name</pre>
<p><b>Syntax</b> <b>Deprecated</b></p> <pre>DROP CONSTRAINT ON (n:Label) ASSERT exists(n.prop)</pre>	<p>Replaced by:</p> <pre>DROP CONSTRAINT name</pre>
<p><b>Syntax</b> <b>Deprecated</b></p> <pre>DROP CONSTRAINT ON ()-[r:Type]-() ASSERT exists (r.prop)</pre>	<p>Replaced by:</p> <pre>DROP CONSTRAINT name</pre>

## Restricted features

Feature	Details
<p><b>Function</b> <b>Restricted</b></p> <pre>length()</pre>	<p>Restricted to only work on paths.</p> <p>See <a href="#">length()</a>.</p>
<p><b>Function</b> <b>Restricted</b></p> <pre>size()</pre>	<p>No longer works for paths. Only works for strings, lists and pattern expressions.</p> <p>See <a href="#">size()</a>.</p>

## Updated features

Feature	Details
<p><b>Syntax</b> <b>Extended</b></p> <pre>CREATE CONSTRAINT [name] ON ...</pre>	<p>The create constraint syntax can now include a name.</p>

## New features

Feature	Details
<p><b>Functionality</b> <b>New</b> Pipelined runtime:</p> <pre>CYPHER runtime=pipelined</pre>	<p>This Neo4j Enterprise Edition only feature involves a new runtime that has many performance enhancements.</p>
<p><b>Functionality</b> <b>New</b> Multi-database administration</p>	<p>New Cypher commands for administering multiple databases.</p>
<p><b>Functionality</b> <b>New</b> Access control</p>	<p>New Cypher commands for administering role-based access control.</p>
<p><b>Functionality</b> <b>New</b> Fine-grained security</p>	<p>New Cypher commands for administering dbms, database, graph and sub-graph access control.</p>
<p><b>Syntax</b> <b>New</b></p> <pre>CREATE INDEX [name] FOR (n:Label) ON (n.prop)</pre>	<p>New syntax for creating indexes, which can include a name.</p>
<p><b>Syntax</b> <b>New</b></p> <pre>DROP INDEX name</pre>	<p><a href="#">New command</a> for dropping an index by name.</p>
<p><b>Syntax</b> <b>New</b></p> <pre>DROP CONSTRAINT name</pre>	<p><a href="#">New command</a> for dropping a constraint by name, no matter the type.</p>
<p><b>Clause</b> <b>New</b></p> <pre>WHERE EXISTS {...}</pre>	<p>Existential sub-queries are sub-clauses used to filter the results of a <b>MATCH</b>, <b>OPTIONAL MATCH</b>, or <b>WITH</b> clause.</p>
<p><b>Clause</b> <b>New</b></p> <pre>USE neo4j</pre>	<p>New clause to specify which graph a query, or query part, is executed against.</p>

## Version 3.5

## Deprecated features

Feature	Details
<p><b>Functionality</b> <b>Deprecated</b> Compiled runtime:</p> <pre>CYPHER runtime=compiled</pre>	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.
<p><b>Function</b> <b>Deprecated</b></p> <pre>extract()</pre>	Replaced by <a href="#">list comprehension</a> .
<p><b>Function</b> <b>Deprecated</b></p> <pre>filter()</pre>	Replaced by <a href="#">list comprehension</a> .

## Version 3.4

Feature	Type	Change	Details
<a href="#">Spatial point types</a>	Functionality	Amendment	A point — irrespective of which Coordinate Reference System is used — can be stored as a property and is able to be backed by an index. Prior to this, a point was a virtual property only.
<a href="#">point() - Cartesian 3D</a>	Function	Added	
<a href="#">point() - WGS 84 3D</a>	Function	Added	
<a href="#">randomUUID()</a>	Function	Added	
<a href="#">Temporal types</a>	Functionality	Added	Supports storing, indexing and working with the following temporal types: <i>Date</i> , <i>Time</i> , <i>LocalTime</i> , <i>DateTime</i> , <i>LocalDateTime</i> and <i>Duration</i> .
<a href="#">Temporal functions</a>	Functionality	Added	Functions allowing for the creation and manipulation of values for each temporal type — <i>Date</i> , <i>Time</i> , <i>LocalTime</i> , <i>DateTime</i> , <i>LocalDateTime</i> and <i>Duration</i> .



Feature	Type	Change	Details
<a href="#">Temporal operators</a>	Functionality	Added	Operators allowing for the manipulation of values for each temporal type — <i>Date</i> , <i>Time</i> , <i>LocalTime</i> , <i>DateTime</i> , <i>LocalDateTime</i> and <i>Duration</i> .
<a href="#">toString()</a>	Function	Extended	Now also allows temporal values as input (i.e. values of type <i>Date</i> , <i>Time</i> , <i>LocalTime</i> , <i>DateTime</i> , <i>LocalDateTime</i> or <i>Duration</i> ).

## Version 3.3

Feature	Type	Change	Details
<code>START</code>	Clause	Removed	As in Cypher 3.2, any queries using the <code>START</code> clause will revert back to Cypher 3.1 <code>planner=rule</code> . However, there are <a href="#">built-in procedures for Neo4j versions 3.3 to 3.5</a> 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 <code>CALL db.index.explicit.searchNodes('my_index', 'email:me*')</code> .
<code>CYPHER runtime=slotted</code> (Faster interpreted runtime)	Functionality	Added	Neo4j Enterprise Edition only.
<code>max()</code> , <code>min()</code>	Function	Extended	Now also supports aggregation over sets containing lists of strings and/or numbers, as well as over sets containing strings, numbers, and lists of strings and/or numbers.

## Version 3.2

Feature	Type	Change	Details
<code>CYPHER planner=rule</code> (Rule planner)	Functionality	Removed	All queries now use the cost planner. Any query prepended thus will fall back to using Cypher 3.1.

Feature	Type	Change	Details
CREATE UNIQUE	Clause	Removed	Running such queries will fall back to using Cypher 3.1 (and use the rule planner).
START	Clause	Removed	Running such queries will fall back to using Cypher 3.1 (and use the rule planner).
<pre>MATCH (n)-[rs*]-() RETURN rs</pre>	Syntax	Deprecated	Replaced by: <pre>MATCH p=(n)-[*]-() RETURN relationships(p) AS rs</pre>
<pre>MATCH (n)-[:A B C {foo: 'bar'}]-() RETURN n</pre>	Syntax	Deprecated	Replaced by: <pre>MATCH (n)-[:A B C {foo: 'bar'}]-() RETURN n</pre>
<pre>MATCH (n)-[x:A B C]-() RETURN n</pre>	Syntax	Deprecated	Replaced by: <pre>MATCH (n)-[x:A B C]-() RETURN n</pre>
<pre>MATCH (n)-[x:A B C*]-() RETURN n</pre>	Syntax	Deprecated	Replaced by: <pre>MATCH (n)-[x:A B C*]-() RETURN n</pre>
User-defined aggregation functions	Functionality	Added	
Composite indexes	Index	Added	
Node Key	Index	Added	Neo4j Enterprise Edition only.
CYPHER runtime=compiled (Compiled runtime)	Functionality	Added	Neo4j Enterprise Edition only.
reverse()	Function	Extended	Now also allows a list as input.
max(), min()	Function	Extended	Now also supports aggregation over a set containing both strings and numbers.

## Version 3.1

Feature	Type	Change	Details
<code>rels()</code>	Function	Deprecated	Replaced by <a href="#">relationships()</a>
<code>toInt()</code>	Function	Deprecated	Replaced by <a href="#">toInteger()</a>
<code>lower()</code>	Function	Deprecated	Replaced by <a href="#">toLowerCase()</a>
<code>upper()</code>	Function	Deprecated	Replaced by <a href="#">toUpperCase()</a>
<code>toBoolean()</code>	Function	Added	
Map projection	Syntax	Added	
Pattern comprehension	Syntax	Added	
User-defined functions	Functionality	Added	
<code>CALL...YIELD...WHERE</code>	Clause	Extended	Records returned by <code>YIELD</code> may be filtered further using <code>WHERE</code>

## Version 3.0

Feature	Type	Change	Details
<code>has()</code>	Function	Removed	Replaced by: <code>exists()</code> See <a href="#">exists()</a> .
<code>str()</code>	Function	Removed	Replaced by: <code>toString()</code> See <a href="#">toString()</a> .
<code>{parameter}</code>	Syntax	Deprecated	Replaced by: <code>\$parameter</code> See <a href="#">Parameters</a> .
<code>properties()</code>	Function	Added	See <a href="#">properties()</a> .

Feature	Type	Change	Details
<code>CALL ... [YIELD]</code>	Clause	Added	See <a href="#">CALL procedure</a> .
<code>point()</code> - Cartesian 2D	Function	Added	
<code>point()</code> - WGS 84 2D	Function	Added	
<code>distance()</code>	Function	Added	See <code>distance()</code> .
User-defined procedures	Functionality	Added	
<code>toString()</code>	Function	Extended	Now also allows Boolean values as input.

## 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 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.



In Neo4j 4.4 the Cypher parser understands some older language features, even if they are no longer supported by the Neo4j kernel. These features result in runtime errors. See the table at [Cypher Version 4.0](#) for the list of affected features.

## 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.

# 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](#)

## Clauses

Clause	Category	Description
<a href="#">CALL</a> [...YIELD]	Reading/Writing	Invoke a procedure deployed in the database.
<a href="#">CALL</a> {...}	Reading/Writing	Evaluates a subquery, typically used for post-union processing or aggregations.
<a href="#">CREATE</a>	Writing	Create nodes and relationships.
<a href="#">CREATE CONSTRAINT</a> [existence] [IF NOT EXISTS] FOR (n:Label) REQUIRE n.property IS NOT NULL [OPTIONS {}]	Schema	Create a constraint ensuring that all nodes with a particular label have a certain property.
<a href="#">CREATE CONSTRAINT</a> [node_key] [IF NOT EXISTS] FOR (n:Label) REQUIRE (n.prop1[, ..., n.propN]) IS NODE KEY [OPTIONS {optionKey: optionValue[, ...]}]	Schema	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.
<a href="#">CREATE CONSTRAINT</a> [existence] [IF NOT EXISTS] FOR ()-["r:REL_TYPE"]-() REQUIRE r.property IS NOT NULL [OPTIONS {}]	Schema	Create a constraint that ensures all relationships with a particular type have a certain property.
<a href="#">CREATE CONSTRAINT</a> [uniqueness] [IF NOT EXISTS] FOR (n:Label) REQUIRE (n.prop1[, ..., n.propN]) IS UNIQUE [OPTIONS {optionKey: optionValue[, ...]}]	Schema	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.

Clause	Category	Description
<code>CREATE [BTREE] INDEX [single] [IF NOT EXISTS] FOR (n:Label) ON (n.property) [OPTIONS {optionKey: optionValue[, ...]}]</code>	Schema	Create an index on all nodes with a particular label and a single property; i.e. create a single-property index.
<code>CREATE [BTREE] INDEX [single] [IF NOT EXISTS] FOR ()-[r:TYPE]"-() ON (r.property) [OPTIONS {optionKey: optionValue[, ...]}]</code>	Schema	Create an index on all relationships with a particular relationship type and a single property; i.e. create a single-property index.
<code>CREATE [BTREE] INDEX [composite] [IF NOT EXISTS] FOR (n:Label) ON (n.prop1, ..., n.propN) [OPTIONS {optionKey: optionValue[, ...]}]</code>	Schema	Create an index on all nodes with a particular label and multiple properties; i.e. create a composite index.
<code>CREATE [BTREE] INDEX [composite] [IF NOT EXISTS] FOR ()-[r:TYPE]"-() ON (r.prop1, ..., r.propN) [OPTIONS {optionKey: optionValue[, ...]}]</code>	Schema	Create an index on all relationships with a particular relationship type and multiple properties; i.e. create a composite index.
<code>CREATE FULLTEXT INDEX [name] [IF NOT EXISTS] FOR (n:Label["" ... "" LabelN]) ON EACH [" n.property[, ..., n.propertyN] "]" [OPTIONS {optionKey: optionValue[, ...]}]</code>	Schema	Create a fulltext index on nodes.
<code>CREATE FULLTEXT INDEX [name] [IF NOT EXISTS] FOR ()-[r:TYPE["" ... "" TYPE_N]"-() ON EACH [" r.property[, ..., r.propertyN] "]" [OPTIONS {optionKey: optionValue[, ...]}]</code>	Schema	Create a fulltext index on relationships.
<code>CREATE LOOKUP INDEX [name] [IF NOT EXISTS] FOR (n) ON EACH labels(n) [OPTIONS {optionKey: optionValue[, ...]}]</code>	Schema	Create an index on all nodes with any label.
<code>CREATE LOOKUP INDEX [name] [IF NOT EXISTS] FOR ()-[r]"-() ON [EACH] type(r) [OPTIONS {optionKey: optionValue[, ...]}]</code>	Schema	Create an index on all relationships with any relationship type.
<code>CREATE TEXT INDEX [name] [IF NOT EXISTS] FOR (n:Label) ON (n.property) [OPTIONS {optionKey: optionValue[, ...]}]</code>	Schema	Create a text index on nodes.
<code>CREATE TEXT INDEX [name] [IF NOT EXISTS] FOR ()-[r:TYPE]"-() ON (r.property) [OPTIONS {optionKey: optionValue[, ...]}]</code>	Schema	Create a text index on relationships.
<code>DELETE</code>	Writing	Delete nodes, relationships or paths. Any node to be deleted must also have all associated relationships explicitly deleted.

Clause	Category	Description
DETACH DELETE	Writing	Delete a node or set of nodes. All associated relationships will automatically be deleted.
DROP CONSTRAINT name [IF EXISTS\]	Schema	Drop a constraint using the name.
DROP INDEX name [IF EXISTS\]	Schema	Drop an index using the name.
FOREACH	Writing	Update data within a list, whether components of a path, or the result of aggregation.
LIMIT	Reading sub-clause	A sub-clause used to constrain the number of rows in the output.
LOAD CSV	Importing data	Use when importing data from CSV files.
MATCH	Reading	Specify the patterns to search for in the database.
MERGE	Reading/Writing	Ensures that a pattern exists in the graph. Either the pattern already exists, or it needs to be created.
ON CREATE	Reading/Writing	Used in conjunction with <b>MERGE</b> , specifying the actions to take if the pattern needs to be created.
ON MATCH	Reading/Writing	Used in conjunction with <b>MERGE</b> , specifying the actions to take if the pattern already exists.
OPTIONAL MATCH	Reading	Specify the patterns to search for in the database while using <b>nulls</b> for missing parts of the pattern.
ORDER BY [ASC[ENDING\]   DESC[ENDING\]]	Reading sub-clause	A sub-clause following <b>RETURN</b> or <b>WITH</b> , specifying that the output should be sorted in either ascending (the default) or descending order.
REMOVE	Writing	Remove properties and labels from nodes and relationships.
RETURN ... [AS\]	Projecting	Defines what to include in the query result set.
SET	Writing	Update labels on nodes and properties on nodes and relationships.
SHOW [ALL UNIQUE NODE [PROPERTY EXIST[ENCE\]] REL[AATIONSHIP] [PROPERTY] EXIST[ENCE\]] PROPERTY EXIST[ENCE\]] NODE KEY] CONSTRAINT[S]]	Schema	List constraints in the database, either all or filtered on type. Also allows <b>WHERE</b> and <b>YIELD</b> clauses.



Clause	Category	Description
<code>SHOW [ALL BTREE FULLTEXT LOOKUP INDEX[ES\]]</code>	Schema	List indexes in the database, either all or filtered on B-tree, fulltext or token lookup indexes. Also allows <b>WHERE</b> and <b>YIELD</b> clauses.
<code>SHOW [ALL BUILT IN USER DEFINED FUNCTION[S] [EXECUTABLE [BY {CURRENT USER username}\]]]</code>	DBMS	List functions, either all or filtered. Available filters are executable by a user or function type (built-in or user-defined). Also allows <b>WHERE</b> and <b>YIELD</b> clauses.
<code>SHOW PROCEDURE[S] [EXECUTABLE [BY {CURRENT USER username}\]]]</code>	DBMS	List procedures, either all or filtered on executable by a user. Also allows <b>WHERE</b> and <b>YIELD</b> clauses.
<code>SHOW TRANSACTION[S] [transaction-id[, ...]\]]</code>	DBMS	List transactions, either all or filtered on ID. Also allows <b>WHERE</b> and <b>YIELD</b> clauses.
<code>SKIP</code>	Reading/Writing	A sub-clause defining from which row to start including the rows in the output.
<code>TERMINATE TRANSACTION[S] transaction-id[, ...]\]]</code>	DBMS	Terminate transactions with the given IDs.
<code>UNION</code>	Set operations	Combines the result of multiple queries. Duplicates are removed.
<code>UNION ALL</code>	Set operations	Combines the result of multiple queries. Duplicates are retained.
<code>UNWIND ... [AS\]</code>	Projecting	Expands a list into a sequence of rows.
<code>USE</code>	Multiple graphs	Determines which graph a query, or query part, is executed against. <a href="#">Fabric</a>
<code>USING INDEX variable:Label(property)</code>	Hint	Index hints are used to specify which index, if any, the planner should use as a starting point.
<code>USING INDEX SEEK variable:Label(property)</code>	Hint	Index seek hint instructs the planner to use an index seek for this clause.
<code>USING JOIN ON variable</code>	Hint	Join hints are used to enforce a join operation at specified points.
<code>USING PERIODIC COMMIT</code>	Hint	This query hint may be used to prevent an out-of-memory error from occurring when importing large amounts of data using <b>LOAD CSV</b> .
<code>USING SCAN variable:Label</code>	Hint	Scan hints are used to force the planner to do a label scan (followed by a filtering operation) instead of using an index.
<code>WITH ... [AS\]</code>	Projecting	Allows query parts to be chained together, piping the results from one to be used as starting points or criteria in the next.

Clause	Category	Description
WHERE	Reading sub-clause	A sub-clause used to add constraints to the patterns in a <b>MATCH</b> or <b>OPTIONAL MATCH</b> clause, or to filter the results of a <b>WITH</b> clause.
WHERE EXISTS {...}	Reading sub-clause	An existential sub-query used to filter the results of a <b>MATCH</b> , <b>OPTIONAL MATCH</b> , or <b>WITH</b> clause.

## Operators

Operator	Category	Description
%	Mathematical	Modulo division.
*	Mathematical	Multiplication.
*	Temporal	Multiplying a duration with a number.
+	Mathematical	Addition.
+	String	Concatenation.
<<query-operators-property, +=>	Property	Property mutation.
+	List	Concatenation
+	Temporal	Adding two durations, or a duration and a temporal instant.
<<query-operators-mathematical, ->	Mathematical	Subtraction or unary minus.
<<query-operators-temporal, ->	Temporal	Subtracting a duration from a temporal instant or from another duration.
.	Map	Static value access by key.
.	Property	Static property access.
/	Mathematical	Division.
/	Temporal	Dividing a duration by a number.
<	Comparison	Less than.
<<query-operators-comparison, <=>	Comparison	Less than or equal to.
<>	Comparison	Inequality.
<<query-operators-comparison, =>	Comparison	Equality.
<<query-operators-property, =>	Property	Property replacement.
=~	String	Regular expression match.
>	Comparison	Greater than.
<<query-operators-comparison, >=>	Comparison	Greater than or equal to.
AND	Boolean	Conjunction.

Operator	Category	Description
CONTAINS	String comparison	Case-sensitive inclusion search.
DISTINCT	Aggregation	Duplicate removal.
ENDS WITH	String comparison	Case-sensitive suffix search.
IN	List	List element existence check.
IS NOT NULL	Comparison	Non- <code>null</code> check.
IS NULL	Comparison	<code>null</code> check.
NOT	Boolean	Negation.
OR	Boolean	Disjunction.
STARTS WITH	String comparison	Case-sensitive prefix search.
XOR	Boolean	Exclusive disjunction.
[ ]	Map	Subscript (dynamic value access by key).
[ ]	Property	Subscript (dynamic property access).
[ ]	List	Subscript (accessing element(s) in a list).
^	Mathematical	Exponentiation.

## Functions

Function	Category	Description
abs()	Numeric	Returns the absolute value of a number.
acos()	Trigonometric	Returns the arccosine of a number in radians.
all()	Predicate	Tests whether the predicate holds for all elements in a list.
any()	Predicate	Tests whether the predicate holds for at least one element in a list.
asin()	Trigonometric	Returns the arcsine of a number in radians.
atan()	Trigonometric	Returns the arctangent of a number in radians.
atan2()	Trigonometric	Returns the arctangent2 of a set of coordinates in radians.
avg()	Aggregating	Returns the average of a set of values.
ceil()	Numeric	Returns the smallest floating point number that is greater than or equal to a number and equal to a mathematical integer.

Function	Category	Description
<code>coalesce()</code>	Scalar	Returns the first non- <code>null</code> value in a list of expressions.
<code>collect()</code>	Aggregating	Returns a list containing the values returned by an expression.
<code>cos()</code>	Trigonometric	Returns the cosine of a number.
<code>cot()</code>	Trigonometric	Returns the cotangent of a number.
<code>count()</code>	Aggregating	Returns the number of values or rows.
<code>date()</code>	Temporal	Returns the current Date.
<code>date({year [, month, day]})</code>	Temporal	Returns a calendar (Year-Month-Day) Date.
<code>date({year [, week, dayOfWeek]})</code>	Temporal	Returns a week (Year-Week-Day) Date.
<code>date({year [, quarter, dayOfQuarter]})</code>	Temporal	Returns a quarter (Year-Quarter-Day) Date.
<code>date({year [, ordinalDay]})</code>	Temporal	Returns an ordinal (Year-Day) Date.
<code>date(string)</code>	Temporal	Returns a Date by parsing a string.
<code>date({map})</code>	Temporal	Returns a Date from a map of another temporal value's components.
<code>date.realtime()</code>	Temporal	Returns the current Date using the <code>realtime</code> clock.
<code>date.statement()</code>	Temporal	Returns the current Date using the <code>statement</code> clock.
<code>date.transaction()</code>	Temporal	Returns the current Date using the <code>transaction</code> clock.
<code>date.truncate()</code>	Temporal	Returns a Date obtained by truncating a value at a specific component boundary. <a href="#">Truncation summary</a> .
<code>datetime()</code>	Temporal	Returns the current DateTime.
<code>datetime({year [, month, day, ...]})</code>	Temporal	Returns a calendar (Year-Month-Day) DateTime.
<code>datetime({year [, week, dayOfWeek, ...]})</code>	Temporal	Returns a week (Year-Week-Day) DateTime.
<code>datetime({year [, quarter, dayOfQuarter, ...]})</code>	Temporal	Returns a quarter (Year-Quarter-Day) DateTime.
<code>datetime({year [, ordinalDay, ...]})</code>	Temporal	Returns an ordinal (Year-Day) DateTime.
<code>datetime(string)</code>	Temporal	Returns a DateTime by parsing a string.
<code>datetime({map})</code>	Temporal	Returns a DateTime from a map of another temporal value's components.

Function	Category	Description
<code>datetime({epochSeconds})</code>	Temporal	Returns a <i>DateTime</i> from a timestamp.
<code>datetime.realtime()</code>	Temporal	Returns the current <i>DateTime</i> using the <i>realtime</i> clock.
<code>datetime.statement()</code>	Temporal	Returns the current <i>DateTime</i> using the <i>statement</i> clock.
<code>datetime.transaction()</code>	Temporal	Returns the current <i>DateTime</i> using the <i>transaction</i> clock.
<code>datetime.truncate()</code>	Temporal	Returns a <i>DateTime</i> obtained by truncating a value at a specific component boundary. <a href="#">Truncation summary</a> .
<code>degrees()</code>	Trigonometric	Converts radians to degrees.
<code>duration({map})</code>	Temporal	Returns a <i>Duration</i> from a map of its components.
<code>duration(string)</code>	Temporal	Returns a <i>Duration</i> by parsing a string.
<code>duration.between()</code>	Temporal	Returns a <i>Duration</i> equal to the difference between two given instants.
<code>duration.inDays()</code>	Temporal	Returns a <i>Duration</i> equal to the difference in whole days or weeks between two given instants.
<code>duration.inMonths()</code>	Temporal	Returns a <i>Duration</i> equal to the difference in whole months, quarters or years between two given instants.
<code>duration.inSeconds()</code>	Temporal	Returns a <i>Duration</i> equal to the difference in seconds and fractions of seconds, or minutes or hours, between two given instants.
<code>e()</code>	Logarithmic	Returns the base of the natural logarithm, <i>e</i> .
<code>endNode()</code>	Scalar	Returns the end node of a relationship.
<code>exists()</code>	Predicate	Returns true if a match for the pattern exists in the graph, or if the specified property exists in the node, relationship or map.
<code>exp()</code>	Logarithmic	Returns $e^n$ , where <i>e</i> is the base of the natural logarithm, and <i>n</i> is the value of the argument expression.
<code>floor()</code>	Numeric	Returns the largest floating point number that is less than or equal to a number and equal to a mathematical integer.
<code>haversin()</code>	Trigonometric	Returns half the versine of a number.

Function	Category	Description
<code>head()</code>	Scalar	Returns the first element in a list.
<code>id()</code>	Scalar	Returns the id of a relationship or node.
<code>isEmpty()</code>	Predicate	Returns true if the given list or map contains no elements or if the given string contains no characters.
<code>keys()</code>	List	Returns a list containing the string representations for all the property names of a node, relationship, or map.
<code>labels()</code>	List	Returns a list containing the string representations for all the labels of a node.
<code>last()</code>	Scalar	Returns the last element in a list.
<code>left()</code>	String	Returns a string containing the specified number of leftmost characters of the original string.
<code>length()</code>	Scalar	Returns the length of a path.
<code>localdatetime()</code>	Temporal	Returns the current <code>LocalDateTime</code> .
<code>localdatetime({year [, month, day, ...\]})</code>	Temporal	Returns a calendar (Year-Month-Day) <code>LocalDateTime</code> .
<code>localdatetime({year [, week, dayOfWeek, ...\]})</code>	Temporal	Returns a week (Year-Week-Day) <code>LocalDateTime</code> .
<code>localdatetime({year [, quarter, dayOfQuarter, ...\]})</code>	Temporal	Returns a quarter (Year-Quarter-Day) <code>DateTime</code> .
<code>localdatetime({year [, ordinalDay, ...\]})</code>	Temporal	Returns an ordinal (Year-Day) <code>LocalDateTime</code> .
<code>localdatetime(string)</code>	Temporal	Returns a <code>LocalDateTime</code> by parsing a string.
<code>localdatetime({map})</code>	Temporal	Returns a <code>LocalDateTime</code> from a map of another temporal value's components.
<code>localdatetime.realtime()</code>	Temporal	Returns the current <code>LocalDateTime</code> using the <code>realtime</code> clock.
<code>localdatetime.statement()</code>	Temporal	Returns the current <code>LocalDateTime</code> using the <code>statement</code> clock.
<code>localdatetime.transaction()</code>	Temporal	Returns the current <code>LocalDateTime</code> using the <code>transaction</code> clock.
<code>localdatetime.truncate()</code>	Temporal	Returns a <code>LocalDateTime</code> obtained by truncating a value at a specific component boundary. <a href="#">Truncation summary</a> .
<code>localtime()</code>	Temporal	Returns the current <code>LocalTime</code> .

Function	Category	Description
<code>localtime({hour [, minute, second, ...\]})</code>	Temporal	Returns a <i>LocalTime</i> with the specified component values.
<code>localtime(string)</code>	Temporal	Returns a <i>LocalTime</i> by parsing a string.
<code>localtime({time [, hour, ...\]})</code>	Temporal	Returns a <i>LocalTime</i> from a map of another temporal value's components.
<code>localtime.realtime()</code>	Temporal	Returns the current <i>LocalTime</i> using the <b>realtime</b> clock.
<code>localtime.statement()</code>	Temporal	Returns the current <i>LocalTime</i> using the <b>statement</b> clock.
<code>localtime.transaction()</code>	Temporal	Returns the current <i>LocalTime</i> using the <b>transaction</b> clock.
<code>localtime.truncate()</code>	Temporal	Returns a <i>LocalTime</i> obtained by truncating a value at a specific component boundary. <a href="#">Truncation summary</a> .
<code>log()</code>	Logarithmic	Returns the natural logarithm of a number.
<code>log10()</code>	Logarithmic	Returns the common logarithm (base 10) of a number.
<code>lTrim()</code>	String	Returns the original string with leading whitespace removed.
<code>max()</code>	Aggregating	Returns the maximum value in a set of values.
<code>min()</code>	Aggregating	Returns the minimum value in a set of values.
<code>nodes()</code>	List	Returns a list containing all the nodes in a path.
<code>none()</code>	Predicate	Returns true if the predicate holds for no element in a list.
<code>percentileCont()</code>	Aggregating	Returns the percentile of the given value over a group using linear interpolation.
<code>percentileDisc()</code>	Aggregating	Returns the nearest value to the given percentile over a group using a rounding method.
<code>pi()</code>	Trigonometric	Returns the mathematical constant <i>pi</i> .
<code>point() - Cartesian 2D</code>	Spatial	Returns a 2D point object, given two coordinate values in the Cartesian coordinate system.
<code>point() - Cartesian 3D</code>	Spatial	Returns a 3D point object, given three coordinate values in the Cartesian coordinate system.

Function	Category	Description
<code>point()</code> - WGS 84 2D	Spatial	Returns a 2D point object, given two coordinate values in the WGS 84 coordinate system.
<code>point()</code> - WGS 84 3D	Spatial	Returns a 3D point object, given three coordinate values in the WGS 84 coordinate system.
<code>point.distance()</code>	Spatial	Returns true if the provided point is within the bounding box defined by the two provided points.
<code>point.withinBBox()</code>	Spatial	Returns a floating point number representing the geodesic distance between any two points in the same CRS.
<code>properties()</code>	Scalar	Returns a map containing all the properties of a node or relationship.
<code>radians()</code>	Trigonometric	Converts degrees to radians.
<code>rand()</code>	Numeric	Returns a random floating point number in the range from 0 (inclusive) to 1 (exclusive); i.e. <code>[0, 1)</code> .
<code>randomUUID()</code>	Scalar	Returns a string value corresponding to a randomly-generated UUID.
<code>range()</code>	List	Returns a list comprising all integer values within a specified range.
<code>reduce()</code>	List	Runs an expression against individual elements of a list, storing the result of the expression in an accumulator.
<code>relationships()</code>	List	Returns a list containing all the relationships in a path.
<code>replace()</code>	String	Returns a string in which all occurrences of a specified string in the original string have been replaced by another (specified) string.
<code>reverse()</code>	List	Returns a list in which the order of all elements in the original list have been reversed.
<code>reverse()</code>	String	Returns a string in which the order of all characters in the original string have been reversed.
<code>right()</code>	String	Returns a string containing the specified number of rightmost characters of the original string.



Function	Category	Description
<code>round()</code>	Numeric	Returns the value of the given number rounded to the nearest integer, with half-way values always rounded up.
<code>round()</code> , with precision	Numeric	Returns the value of the given number rounded with the specified precision, with half-values always being rounded up.
<code>round()</code> , with precision and rounding mode	Numeric	Returns the value of the given number rounded with the specified precision and the specified rounding mode.
<code>rTrim()</code>	String	Returns the original string with trailing whitespace removed.
<code>sign()</code>	Numeric	Returns the signum of a number: <code>0</code> if the number is <code>0</code> , <code>-1</code> for any negative number, and <code>1</code> for any positive number.
<code>sin()</code>	Trigonometric	Returns the sine of a number.
<code>single()</code>	Predicate	Returns true if the predicate holds for exactly one of the elements in a list.
<code>size()</code>	Scalar	Returns the number of items in a list.
<code>size()</code> applied to pattern comprehension	Scalar	Returns the number of paths matching the pattern comprehension.
<code>size()</code> applied to string	Scalar	Returns the number of Unicode characters in a string.
<code>split()</code>	String	Returns a list of strings resulting from the splitting of the original string around matches of the given delimiter.
<code>sqrt()</code>	Logarithmic	Returns the square root of a number.
<code>startNode()</code>	Scalar	Returns the start node of a relationship.
<code>stDev()</code>	Aggregating	Returns the standard deviation for the given value over a group for a sample of a population.
<code>stDevP()</code>	Aggregating	Returns the standard deviation for the given value over a group for an entire population.
<code>substring()</code>	String	Returns a substring of the original string, beginning with a 0-based index start and length.
<code>sum()</code>	Aggregating	Returns the sum of a set of numeric values.
<code>tail()</code>	List	Returns all but the first element in a list.
<code>tan()</code>	Trigonometric	Returns the tangent of a number.

Function	Category	Description
<code>time()</code>	Temporal	Returns the current <i>Time</i> .
<code>time({hour [, minute, ...]})</code>	Temporal	Returns a <i>Time</i> with the specified component values.
<code>time(string)</code>	Temporal	Returns a <i>Time</i> by parsing a string.
<code>time({time [, hour, ..., timezone]})</code>	Temporal	Returns a <i>Time</i> from a map of another temporal value's components.
<code>time.realtime()</code>	Temporal	Returns the current <i>Time</i> using the <b>realtime</b> clock.
<code>time.statement()</code>	Temporal	Returns the current <i>Time</i> using the <b>statement</b> clock.
<code>time.transaction()</code>	Temporal	Returns the current <i>Time</i> using the <b>transaction</b> clock.
<code>time.truncate()</code>	Temporal	Returns a <i>Time</i> obtained by truncating a value at a specific component boundary. <a href="#">Truncation summary</a> .
<code>timestamp()</code>	Scalar	Returns the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC.
<code>toBoolean()</code>	Scalar	Converts a string value to a boolean value.
<code>toFloat()</code>	Scalar	Converts an integer or string value to a floating point number.
<code>toInteger()</code>	Scalar	Converts a floating point or string value to an integer value.
<code>toLower()</code>	String	Returns the original string in lowercase.
<code>toString()</code>	String	Converts an integer, float, boolean or temporal (i.e. <i>Date</i> , <i>Time</i> , <i>LocalTime</i> , <i>DateTime</i> , <i>LocalDateTime</i> or <i>Duration</i> ) value to a string.
<code>toUpper()</code>	String	Returns the original string in uppercase.
<code>trim()</code>	String	Returns the original string with leading and trailing whitespace removed.
<code>type()</code>	Scalar	Returns the string representation of the relationship type.

## Expressions

Name	Description
<a href="#">CASE Expression</a>	A generic conditional expression, similar to if/else statements available in other languages.

## Cypher query options

Name	Type	Description
<code>CYPHER \$version query</code>	Version	This will force 'query' to use Neo4j Cypher \$version.
<code>CYPHER runtime=interpreted query</code>	Runtime	This will force the query planner to use the interpreted runtime. This is the only option in Neo4j Community Edition.
<code>CYPHER runtime=slotted query</code>	Runtime	This will cause the query planner to use the slotted runtime. This is only available in Neo4j Enterprise Edition.
<code>CYPHER runtime=pipelined query</code>	Runtime	This will cause the query planner to use the pipelined runtime if it supports 'query'. This is only available in Neo4j Enterprise Edition.

## Administrative commands

The following commands are only executable against the `system` database:

Command	Admin category	Description
<code>ALTER ALIAS ... [IF EXISTS] SET DATABASE ...]</code>	Database alias	Modifies a database alias.
<code>ALTER CURRENT USER SET PASSWORD FROM ... TO</code>	User and role	Change the password of the user that is currently logged in.
<code>ALTER DATABASE ... [IF EXISTS] SET ACCESS {READ ONLY   READ WRITE}}</code>	Database	Modifies the database access mode.
<code>ALTER USER ... [IF EXISTS] [SET [PLAINTEXT   ENCRYPTED] PASSWORD {password [CHANGE [NOT] REQUIRED]   CHANGE [NOT] REQUIRED}] [SET STATUS {ACTIVE   SUSPENDED}] [SET HOME DATABASE name] [REMOVE HOME DATABASE]</code>	User and role	Changes a user account. Changes can include setting a new password, setting the account status, setting or removing home database and enabling that the user should change the password upon next login.
<code>CREATE [OR REPLACE] ALIAS ... [IF NOT EXISTS] FOR DATABASE ...]</code>	Database alias	Creates a new database alias.
<code>CREATE [OR REPLACE] DATABASE ... [IF NOT EXISTS] [OPTIONS {optionKey: optionValue[, ...]}] [WAIT [n [SEC[OND[S]]]] NOWAIT]</code>	Database	Creates a new database.
<code>CREATE [OR REPLACE] ROLE ... [IF NOT EXISTS] [AS COPY OF]</code>	User and role	Creates new roles.

Command	Admin category	Description
<code>CREATE [OR REPLACE] USER ... [IF NOT EXISTS] SET [PLAINTEXT   ENCRYPTED] PASSWORD ... [[SET PASSWORD] CHANGE [NOT] REQUIRED] [SET STATUS {ACTIVE   SUSPENDED}] [SET HOME DATABASE name]</code>	User and role	Creates a new user and sets the password for the new account. Optionally the account status and home database can also be set and if the user should change the password upon first login.
<code>DENY ... ON DATABASE ... TO</code>	Privilege	Denies a database or schema privilege to one or multiple roles.
<code>DENY ... ON DBMS TO</code>	Privilege	Denies a DBMS privilege to one or multiple roles.
<code>DENY ... ON GRAPH ... [NODES   RELATIONSHIPS   ELEMENTS] ... TO</code>	Privilege	Denies a graph privilege for one or multiple specified elements to one or multiple roles.
<code>DROP ALIAS ... [IF EXISTS] FOR DATABASE</code>	Database alias	Deletes a specified database alias.
<code>DROP DATABASE ... [IF EXISTS] [DUMP DATA   DESTROY DATA]</code>	Database	Deletes a specified database.
<code>DROP ROLE ... [IF EXISTS]</code>	User and role	Deletes a specified role.
<code>DROP USER ... [IF EXISTS]</code>	User and role	Deletes a specified user.
<code>GRANT ... ON DATABASE ... TO</code>	Privilege	Assigns a database or schema privilege to one or multiple roles.
<code>GRANT ... ON DBMS TO</code>	Privilege	Assigns a DBMS privilege to one or multiple roles.
<code>GRANT ... ON GRAPH ... [NODES   RELATIONSHIPS   ELEMENTS] ... TO</code>	Privilege	Assigns a graph privilege for one or multiple specified elements to one or multiple roles.
<code>GRANT ROLE[S] ... TO</code>	User and role	Assigns one or multiple roles to one or multiple users.
<code>RENAME ROLE ... [IF EXISTS] TO ...</code>	User and role	Changes the name of a role.
<code>RENAME USER ... [IF EXISTS] TO ...</code>	User and role	Changes the name of a user.
<code>REVOKE [GRANT   DENY] ... ON DATABASE ... FROM</code>	Privilege	Removes a database or schema privilege from one or multiple roles.
<code>REVOKE [GRANT   DENY] ... ON DBMS FROM</code>	Privilege	Removes a DBMS privilege from one or multiple roles.
<code>REVOKE [GRANT   DENY] ... ON GRAPH ... [NODES   RELATIONSHIPS   ELEMENTS] ... FROM</code>	Privilege	Removes a graph privilege for one or multiple specified elements from one or multiple roles.
<code>REVOKE ROLE[S] ... FROM</code>	User and role	Removes one or multiple roles from one or multiple users.

Command	Admin category	Description
<code>SHOW ALIASES FOR DATABASE</code>	Database alias	Returns information about all aliases, optionally including driver settings.
<code>SHOW [ALL   POPULATED\] ROLES [WITH USERS\]]</code>	User and role	Returns information about all or populated roles, optionally including the assigned users.
<code>SHOW DATABASE</code>	Database	Returns information about a specified database.
<code>SHOW DATABASES</code>	Database	Returns information about all databases.
<code>SHOW DEFAULT DATABASE</code>	Database	Returns information about the default database.
<code>SHOW HOME DATABASE</code>	Database	Returns information about the current users home database.
<code>SHOW [ROLE ...   USER ...   ALL \] PRIVILEGES [AS [REVOKE] COMMAND[S\]]]</code>	Privilege	Returns information about role, user or all privileges.
<code>SHOW USERS</code>	User and role	Returns information about all users.
<code>START DATABASE</code>	Database	Starts up a specified database.
<code>STOP DATABASE</code>	Database	Stops a specified database.

## Privilege Actions

Name	Category	Description
<code>ACCESS</code>	Database	Determines whether a user can access a specific database.
<code>ALL DATABASE PRIVILEGES</code>	Database and schema	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.
<code>ALL DBMS PRIVILEGES</code>	DBMS	Determines whether a user is allowed to perform role, user, database and privilege management.
<code>ALL GRAPH PRIVILEGES</code>	GRAPH	Determines whether a user is allowed to perform reads and writes.
<code>ALTER ALIAS</code>	DBMS	Determines whether the user can modify aliases.
<code>ALTER DATABASE</code>	DBMS	Determines whether the user can modify databases and aliases.
<code>ALTER USER</code>	DBMS	Determines whether the user can modify users.

Name	Category	Description
ASSIGN PRIVILEGE	DBMS	Determines whether the user can assign privileges using the <b>GRANT</b> and <b>DENY</b> commands.
ASSIGN ROLE	DBMS	Determines whether the user can grant roles.
CONSTRAINT MANAGEMENT	Schema	Determines whether a user is allowed to create, drop, and list constraints on a specific database.
CREATE	GRAPH	Determines whether the user can create a new element (node, relationship or both).
CREATE ALIAS	DBMS	Determines whether the user can create new aliases.
CREATE CONSTRAINT	Schema	Determines whether a user is allowed to create constraints on a specific database.
CREATE DATABASE	DBMS	Determines whether the user can create new databases and aliases.
CREATE INDEX	Schema	Determines whether a user is allowed to create indexes on a specific database.
CREATE NEW NODE LABEL	Schema	Determines whether a user is allowed to create new node labels on a specific database.
CREATE NEW PROPERTY NAME	Schema	Determines whether a user is allowed to create new property names on a specific database.
CREATE NEW RELATIONSHIP TYPE	Schema	Determines whether a user is allowed to create new relationship types on a specific database.
CREATE ROLE	DBMS	Determines whether the user can create new roles.
CREATE USER	DBMS	Determines whether the user can create new users.
ALIAS MANAGEMENT	DBMS	Determines whether the user can create, delete, modify and list aliases.
DATABASE MANAGEMENT	DBMS	Determines whether the user can create, delete, and modify databases and aliases.
DELETE	GRAPH	Determines whether the user can delete an element (node, relationship or both).
DROP ALIAS	DBMS	Determines whether the user can delete aliases.

Name	Category	Description
DROP CONSTRAINT	Schema	Determines whether a user is allowed to drop constraints on a specific database.
DROP DATABASE	DBMS	Determines whether the user can delete databases and aliases.
DROP INDEX	Schema	Determines whether a user is allowed to drop indexes on a specific database.
DROP ROLE	DBMS	Determines whether the user can delete roles.
DROP USER	DBMS	Determines whether the user can delete users.
EXECUTE ADMIN PROCEDURE	DBMS	Determines whether the user can execute admin procedures.
EXECUTE BOOSTED FUNCTION	DBMS	Determines whether the user can execute functions with elevated privileges.
EXECUTE BOOSTED PROCEDURE	DBMS	Determines whether the user can execute procedures with elevated privileges.
EXECUTE FUNCTION	DBMS	Determines whether the user can execute functions.
EXECUTE PROCEDURE	DBMS	Determines whether the user can execute procedures.
IMPERSONATE	DBMS	Determines whether a user can impersonate another one and assume their privileges.
INDEX MANAGEMENT	Schema	Determines whether a user is allowed to create, drop, and list indexes on a specific database.
MATCH	GRAPH	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.
MERGE	GRAPH	Determines whether the user can find, read, create and set properties on an element (node, relationship or both).
NAME MANAGEMENT	Schema	Determines whether a user is allowed to create new labels, types and property names on a specific database.
PRIVILEGE MANAGEMENT	DBMS	Determines whether the user can show, assign and remove privileges.

Name	Category	Description
READ	GRAPH	Determines whether the properties of an element (node, relationship or both) can be read while executing queries on the specified graph.
REMOVE LABEL	GRAPH	Determines whether the user can remove a label from a node using the <b>REMOVE</b> clause.
REMOVE PRIVILEGE	DBMS	Determines whether the user can remove privileges using the <b>REVOKE</b> command.
REMOVE ROLE	DBMS	Determines whether the user can revoke roles.
RENAME ROLE	DBMS	Determines whether the user can rename roles.
RENAME USER	DBMS	Determines whether the user can rename users.
ROLE MANAGEMENT	DBMS	Determines whether the user can create, drop, grant, revoke and show roles.
SET DATABASE ACCESS	DBMS	Determines whether the user can modify the database access mode.
SET LABEL	GRAPH	Determines whether the user can set a label to a node using the SET clause.
SET PASSWORDS	DBMS	Determines whether the user can modify users' passwords and whether those passwords must be changed upon first login.
SET PROPERTY	GRAPH	Determines whether the user can set a property to an element (node, relationship or both) using the SET clause.
SET USER HOME DATABASE	DBMS	Determines whether the user can modify the home database of users.
SET USER STATUS	DBMS	Determines whether the user can modify the account status of users.
SHOW ALIAS	DBMS	Determines whether the user is allowed to list aliases.
SHOW CONSTRAINT	Schema	Determines whether the user is allowed to list constraints.
SHOW INDEX	Schema	Determines whether the user is allowed to list indexes.



Name	Category	Description
SHOW PRIVILEGE	DBMS	Determines whether the user can get information about privileges assigned to users and roles.
SHOW ROLE	DBMS	Determines whether the user can get information about existing and assigned roles.
SHOW TRANSACTION	Database	Determines whether a user is allowed to list transactions and queries.
SHOW USER	DBMS	Determines whether the user can get information about existing users.
START	Database	Determines whether a user can start up a specific database.
STOP	Database	Determines whether a user can stop a specific running database.
TERMINATE TRANSACTION	Database	Determines whether a user is allowed to end running transactions and queries.
TRANSACTION MANAGEMENT	Database	Determines whether a user is allowed to list and end running transactions and queries.
TRAVERSE	GRAPH	Determines whether an element (node, relationship or both) can be found and traversed while executing queries on the specified graph.
USER MANAGEMENT	DBMS	Determines whether the user can create, drop, modify and show users.
WRITE	GRAPH	Determines whether the user can execute write operations on the specified graph.

# Appendix A: Cypher styleguide

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](#).

## 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**.

## Indentation and line breaks

- Start a new clause on a new line.

Bad

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

Good

```
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
```

# Casing

- Write keywords in upper case.

Bad

```
match (p:Person)
where p.name starts with 'Ma'
return p.name
```

Good

```
MATCH (p:Person)
WHERE p.name STARTS WITH 'Ma'
RETURN p.name
```

- Write the value `null` in lower case.

Bad

```
WITH NULL AS n1, Null AS n2
RETURN n1 IS NULL AND n2 IS NOT NULL
```

Good

```
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

```
WITH TRUE AS b1, False AS b2
RETURN b1 AND b2
```

Good

```
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

```
CREATE (N {Prop: 0})
WITH RAND() AS Rand, $pArAm AS MAP
RETURN Rand, MAP.property_key, Count(N)
```

Good

```
CREATE (n {prop: 0})  
WITH rand() AS rand, $param AS map  
RETURN rand, map.propertyKey, count(n)
```

## 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

```
WITH { key1 : 'value' ,key2 : 42 } AS map  
RETURN map
```

Good

```
WITH {key1: 'value', key2: 42} AS map  
RETURN map
```

- One space between label/type predicates and property predicates in patterns.

Bad

```
MATCH (p:Person{property: -1})-[:KNOWS {since: 2016}]->()  
RETURN p.name
```

Good

```
MATCH (p:Person {property: -1})-[:KNOWS {since: 2016}]->()  
RETURN p.name
```

- No space in patterns.

Bad

```
MATCH (:Person) --> (:Vehicle)  
RETURN count(*)
```

Good

```
MATCH (:Person)-->(:Vehicle)  
RETURN count(*)
```

- Use a wrapping space around operators.

Bad

```
MATCH p=(s)-->(e)
WHERE s.name<>e.name
RETURN length(p)
```

Good

```
MATCH p = (s)-->(e)
WHERE s.name <> e.name
RETURN length(p)
```

- No space in label predicates.

Bad

```
MATCH (person : Person : Owner )
RETURN person.name
```

Good

```
MATCH (person:Person:Owner)
RETURN person.name
```

- Use a space after each comma in lists and enumerations.

Bad

```
MATCH (),()
WITH ['a','b',3.14] AS list
RETURN list,2,3,4
```

Good

```
MATCH (), ()
WITH ['a', 'b', 3.14] AS list
RETURN list, 2, 3, 4
```

- No padding space within function call parentheses.

Bad

```
RETURN split( 'original', 'i' )
```

Good

```
RETURN split('original', 'i')
```

- Use padding space within simple subquery expressions.

Bad

```
MATCH (a:A)
WHERE EXISTS {(a)-->(b:B)}
RETURN a.prop
```

Good

```
MATCH (a:A)
WHERE EXISTS { (a)-->(b:B) }
RETURN a.prop
```

## 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

```
MATCH (kate:Person {name: 'Kate'})-[r:LIKES]-(c:Car)
RETURN c.type
```

Good

```
MATCH (:Person {name: 'Kate'})-[r:LIKES]-(c:Car)
RETURN c.type
```

- 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.

Bad

```
MATCH ()-->(vehicle:Car)-->(manufacturer:Company)
WHERE manufacturer.foundedYear < 2000
RETURN vehicle.mileage
```

Good

```
MATCH (manufacturer:Company)<--(vehicle:Car)<--()
WHERE manufacturer.foundedYear < 2000
RETURN vehicle.mileage
```

- Keep anchor nodes at the beginning of the **MATCH** clause.

Bad

```
MATCH (:Person)-->(vehicle:Car)-->(manufacturer:Company)
WHERE manufacturer.foundedYear < 2000
RETURN vehicle.mileage
```

Good

```
MATCH (manufacturer:Company)<--(vehicle:Car)<--(:Person)
WHERE manufacturer.foundedYear < 2000
RETURN vehicle.mileage
```

- Prefer outgoing (left to right) pattern relationships to incoming pattern relationships.

Bad

```
MATCH (:Country)-->(:Company)<--(vehicle:Car)<--(:Person)
RETURN vehicle.mileage
```

Good

```
MATCH (:Person)-->(vehicle:Car)-->(:Company)<--(:Country)
RETURN vehicle.mileage
```

## Meta-characters

- Use single quotes, `'`, for literal string values.

Bad

```
RETURN "Cypher"
```

Good

```
RETURN 'Cypher'
```

- 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>.