



The Neo4j Cypher Manual v5

Table of Contents

Documentation updates for Neo4j 5	2
Introduction	5
What is Cypher?	5
Neo4j databases and graphs	7
Querying, updating and administering	9
Transactions	10
Cypher path matching	12
Clause composition	15
Syntax	33
Values and types	35
Naming rules and recommendations	37
Expressions	39
Variables	65
Reserved keywords	66
Parameters	69
Operators	73
Comments	90
Patterns	91
Temporal (Date/Time) values	96
Spatial values	117
Lists	123
Maps	127
Working with <code>null</code>	129
Clauses	132
Administration clauses	132
Importing data	132
Listing functions and procedures	132
Multiple graphs	132
Projecting clauses	133
Reading clauses	133
Reading hints	133
Reading sub-clauses	133
Reading/Writing clauses	134
Set operations	134
Subquery clauses	134
Transaction Commands	135
Writing clauses	135
MATCH	135

OPTIONAL MATCH.....	153
RETURN.....	155
WITH	159
UNWIND	163
WHERE	167
ORDER BY.....	183
SKIP	187
LIMIT.....	188
CREATE.....	191
DELETE	195
SET	198
REMOVE	205
FOREACH	206
MERGE.....	207
CALL {} (subquery)	218
CALL procedure	229
UNION	235
USE	236
LOAD CSV.....	238
SHOW FUNCTIONS	245
SHOW PROCEDURES.....	250
Functions	258
Predicate functions	270
Scalar functions	278
Aggregating functions	299
List functions.....	318
Mathematical functions - numeric	329
Mathematical functions - logarithmic	341
Mathematical functions - trigonometric	346
String functions	358
Temporal functions - instant types	370
Temporal functions - duration.....	440
Spatial functions.....	449
LOAD CSV functions.....	460
Graph functions	461
User-defined functions.....	463
Indexes for search performance	466
Indexes (types and limitations)	466
Syntax.....	467
Composite index limitations.....	470
CREATE INDEX	471

SHOW INDEXES	485
SHOW INDEXES	486
DROP INDEX	487
DROP INDEX	487
Full-text search index	489
Full-text search procedures.....	490
Create and configure full-text indexes	490
Query full-text indexes	493
Handling of Text Array properties	495
Drop full-text indexes	496
Constraints.....	498
Types of constraint	498
Implications on indexes	499
Syntax.....	499
Examples	503
Database management.....	522
Listing databases.....	523
Creating databases Enterprise edition	530
Altering databases Enterprise edition	534
Stopping databases Enterprise edition	536
Starting databases Enterprise edition	537
Deleting databases Enterprise edition	538
Wait options Enterprise edition	540
Database alias management	542
Listing database aliases Enterprise edition	545
Creating database aliases Enterprise edition	548
Create database aliases in composite databases Enterprise edition	555
Altering database aliases.....	557
Deleting database aliases Enterprise edition	561
Access control.....	565
Syntax summaries	565
Managing users	566
Managing roles.....	578
Managing privileges	587
Managing servers.....	603
Built-in roles and privileges.....	609
Read privileges.....	616
Write privileges	619
Database administration	624
DBMS administration	636
Limitations	672

Immutable privileges	677
Query tuning	679
Cypher query options	679
Profile a query.....	684
The use of indexes.....	685
Basic query tuning example.....	709
Advanced query tuning example	721
Planner hints and the USING keyword	739
Execution plans	757
Database hits	759
Execution plan operators	760
Execution plan operators in detail	770
Shortest path planning.....	883
Deprecations, additions, and compatibility	889
Version 5.3	889
Version 5.2	889
Version 5.1	890
Version 5.0	891
Version 4.4	904
Version 4.3	911
Version 4.2	917
Version 4.1.3.....	920
Version 4.1	920
Version 4.0	922
Version 3.5	927
Version 3.4	927
Version 3.3	928
Version 3.2	929
Version 3.1	930
Version 3.0	930
Glossary of keywords	931
Clauses.....	931
Operators.....	935
Functions	936
Expressions	944
Cypher query options	944
Administrative commands.....	944
Privilege Actions	947
Appendix A: Cypher styleguide.....	952
General recommendations.....	952
Indentation and line breaks	952

Casing.....	954
Spacing	955
Patterns	957
Meta-characters.....	958

Cypher is Neo4j's graph query language that allows users to store and retrieve data from the graph database. It is a declarative, SQL-inspired language for describing visual patterns in graphs. The syntax provides a visual and logical way to match patterns of nodes and relationships in the graph.

Documentation updates for Neo4j 5

Neo4j 5 includes a number of new features and updates. A highlight of these include:

- Cypher syntax improvements with Graph Pattern Matching (relationships and labels):
 - In `MATCH` clauses, `WHERE` can be placed inside a relationship pattern to filter relationships.
 - In `MATCH` clauses, nodes and relationships can be filtered using more sophisticated label (type) expressions.
 - Simpler alternative syntax to navigate and traverse graphs using the following operators:
 - `&`: logical AND
 - `|`: logical OR
 - `!`: logical NOT
 - `%`: a "wildcard", meaning "any label" (in Cypher this translates to `size(labels(n)) > 0`).

For more information, see the [section on Label expressions](#) and in the [WHERE clause](#).

- New `elementID` for graph objects:

New IDs are introduced to uniquely identify graph elements in Neo4j databases. Node ID will exist with each release of Neo4j 5.

For more information, see [elementId\(\)](#).

- Composite databases.

Composite databases allow queries that access multiple graphs at once. You can create, update, and remove configurations without a restart, whether the database is within the same cluster, or hosted remotely.

For more information on composite databases, and how to create composite databases, see [Operations Manual → Composite databases](#), and [Creating composite databases](#).

- Immutable privileges.

Immutable privileges are useful for restricting the actions of users who themselves are able to administer privileges.

For more information, see [Immutable privileges](#).

- `Execute` and `ExecuteBoosted` privilege.

The permissions for the execution of admin procedures have been refreshed; these two privileges are now hierarchically related.

For more information, see [the EXECUTE PROCEDURE privilege](#) and [the EXECUTE BOOSTED PROCEDURE privilege](#).

- `EXISTS` and `COUNT` are now both expressions.

For more information, see [Subquery expressions](#).

- SHOW and TERMINATE TRANSACTIONS improvements.

You can now combine these two commands in the same query. The ability to yield and filter the output from TERMINATE TRANSACTIONS has been added.

For more information, see [Updated features list](#).

- Changes to Neo4j indexes:

- The B-tree index type has been removed.
- New Range and Point index types are available now.
- Faster Text index provider for ENDS WITH and CONTAINS queries is introduced.
- Full-text indexes can now index lists of strings.

For more information, see [new index types](#).

© 2023 Neo4j, Inc.

Documentation license: [Creative Commons 4.0](#)

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>.

Introduction

What is Cypher?

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

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

- 
- Cypher keywords are not case-sensitive.
 - Cypher is case-sensitive for variables.
 - There are special naming rules for database names.
 - There are special naming rules for database aliases.

Structure

Cypher borrows its structure from SQL — queries are built up using various clauses.

Clauses are chained together, and they feed intermediate result sets between each other. For example, the matching variables from one `MATCH` clause will be the context that the next clause exists in.

The query language is comprised of several distinct clauses. These are discussed in more detail in the chapter on [Clauses](#).

The following are a few examples of clauses used to read from the graph:

- **MATCH**: The graph pattern to match. This is the most common way to get data from the graph.
- **WHERE**: Not a clause in its own right, but rather part of `MATCH`, `OPTIONAL MATCH` and `WITH`. Adds constraints to a pattern, or filters the intermediate result passing through `WITH`.
- **RETURN**: What to return.

And these are examples of clauses that are used to update the graph:

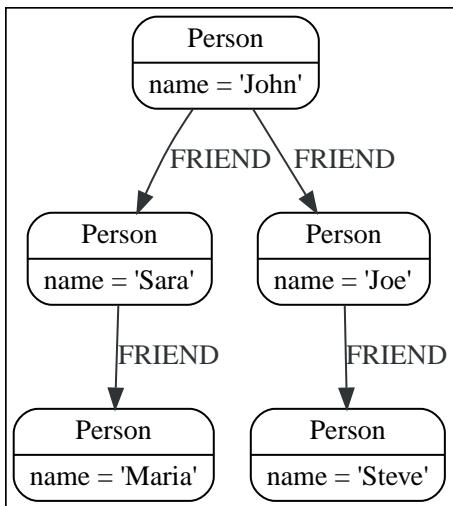
- **CREATE** (and **DELETE**): Create (and delete) nodes and relationships.
- **SET** (and **REMOVE**): Set values to properties and add labels on nodes using `SET` and use `REMOVE` to remove them.
- **MERGE**: Match existing or create new nodes and patterns. This is especially useful together with property uniqueness constraints.

Example 1. Cypher Query

Let's see **MATCH** and **RETURN** in action.

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

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



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

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

Resulting in:

```
Rows: 2
+-----+
| john.name | fof.name |
+-----+
| 'John'    | 'Maria'   |
| 'John'    | 'Steve'   |
+-----+
```

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

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

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

Resulting in:

Rows: 2	
user.name	follower.name
'John'	'Sara'
'Joe'	'Steve'

Neo4j databases and graphs

This section describes databases and graphs in Neo4j.

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

DBMS

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

Graph

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

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

Composite databases can contain multiple graphs, by means of aliases to other databases. Queries submitted to composite databases may refer to multiple graphs within the same query.

For more information, see [Operations manual → Composite databases](#).

Database

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

Most of the time Cypher queries are [reading or updating queries](#), which are run against a graph. There are also administrative commands that apply to a database, or to the entire DBMS. Administrative commands cannot be run in a session connected to a normal user database, but instead need to be run within a session connected to the system database. Administrative commands execute on the system database. If an administrative command is submitted to a user database, it is rerouted to the system database.

The system database and the default database

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

A fresh installation of Neo4j includes two databases:

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

For more information about the system database, see the sections on [Database management](#) and [Access control](#).

Different editions of Neo4j

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

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

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

Limited Support Features

Some elements of Cypher do not work in all deployments of Neo4j.

Specific labels are added to the documentation to highlight these cases.

Description	Label
This feature has been deprecated and will be removed or replaced in the future.	Deprecated
This feature only works in the enterprise edition of Neo4j.	Enterprise edition

Querying, updating and administering

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

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

All these various options are described in more detail in later sections, but it is worth summarizing a few key points first.

The structure of administrative queries

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

The structure of update queries

If you read from the graph and then update the graph, your query implicitly has two parts — the reading is the first part, and the writing is the second part.



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

If your query only performs reads, Cypher will not actually match the pattern until you ask for the results. In an updating query, the semantics are that all the reading will be done before any writing is performed.

The only pattern where the query parts are implicit is when you first read and then write — any other order and you have to be explicit about your query parts. The parts are separated using the `WITH` statement. `WITH` is like an event horizon — it's a barrier between a plan and the finished execution of that plan.

When you want to filter using aggregated data, you have to chain together two reading query parts — the first one does the aggregating, and the second filters on the results coming from the first one.

```
MATCH (n {name: 'John'})-[:FRIEND]-(friend)
WITH n, count(friend) AS friendsCount
WHERE friendsCount > 3
RETURN n, friendsCount
```

Using `WITH`, you specify how you want the aggregation to happen, and that the aggregation has to be finished before Cypher can start filtering.

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

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

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

Returning data

Any query can return data. If a query only reads, it has to return data. If a read-query doesn't return any data, it serves no purpose, and is therefore not a valid Cypher query. Queries that update the graph don't have to return anything, but they can.

After all the parts of the query comes one final `RETURN` clause. `RETURN` is not part of any query part—it is a period symbol at the end of a query. The `RETURN` clause has three sub-clauses that come with it: `SKIP/LIMIT` and `ORDER BY`.

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

Transactions

This section describes how Cypher queries work with database transactions.

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

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



A query that makes a large number of updates consequently uses large amounts of memory since the transaction holds changes in memory. For memory configuration in Neo4j, see the [Neo4j Operations Manual → Memory configuration](#).

Transactions can be either **explicit** or **implicit**.

- **Explicit transactions:**
 - Are opened by the user.
 - Can execute multiple Cypher queries in sequence.
 - Are committed, or rolled back, by the user.
- **Implicit transactions**, sometimes called auto-commit transactions or `:auto` transactions:
 - Are opened automatically.
 - Can execute a single Cypher query.

- Are committed automatically when the query finishes successfully.

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

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

DBMS transactions have the following limitations:

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

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

Cypher path matching

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

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



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

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

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

Homomorphism

No constraints for path matching.

Node isomorphism

The same node cannot be returned more than once for each path matching record.

Relationship isomorphism

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

Homomorphism

Constraints: No constraints for path matching.

Example 2. Homomorphism

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

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

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

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

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

Node isomorphism

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

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

Example 3. Node isomorphism

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

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

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

Relationship isomorphism

Constraints: The same relationship cannot be returned more than once for each path matching record.

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

Example 4. Relationship isomorphism

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

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

This will return the two resulting records `[a, b]`, as well as `[b, a]`.

Cypher path matching example

Cypher makes use of relationship isomorphism for path matching.

Example 5. Friend of friends

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

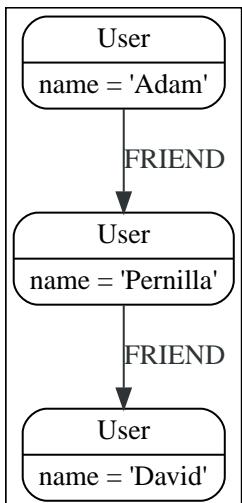
To demonstrate this, let's create a few nodes and relationships:

Query 1, create data.

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

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

Which gives us the following graph:



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

Query 2, friend of friends of Adam.

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

```
Rows: 1
```

```
+-----+
| fofName |
+-----+
| "David" |
+-----+
```

In this query, Cypher makes sure to not return matches where the pattern relationships `r1` and `r2` point to the same graph relationship.

This is however not always desired. If the query should return the user, it is possible to spread the

matching over multiple `MATCH` clauses, like so:

Query 3, multiple `MATCH` clauses.

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

Rows: 2

fofName
"David"
"Adam"

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

Query 4, equivalent to query 2.

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

Here, the `MATCH` clause has a single pattern with two paths, while the previous query has two distinct patterns.

Rows: 1

fofName
"David"

Clause composition

This section describes the semantics of Cypher when composing different read and write clauses.

A query is made up from several clauses chained together. These are discussed in more detail in the chapter on [Clauses](#).

The semantics of a whole query is defined by the semantics of its clauses. Each clause has as input the state of the graph and a table of intermediate results consisting of the current variables. 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 first clause takes as input the state of the graph before the query and an empty table of intermediate results. The output of the last clause is the result of the query.

Example 6. Table of intermediate results between read clauses

The following example graph is used throughout this section.

```
digraph L { node [shape=record style=rounded];
  N0 [
    label = "{Person|name = \\'John\\\1}"
  ]
  N0 -> N3 [
    color = "#2e3436"
    fontcolor = "#2e3436"
    label = "FRIEND\n"
  ]
  N0 -> N1 [
    color = "#2e3436"
    fontcolor = "#2e3436"
    label = "FRIEND\n"
  ]
  N1 [
    label = "{Person|name = \\'Joe\\\1}"
  ]
  N1 -> N2 [
    color = "#2e3436"
    fontcolor = "#2e3436"
    label = "FRIEND\n"
  ]
  N2 [
    label = "{Person|name = \\'Steve\\\1}"
  ]
  N3 [
    label = "{Person|name = \\'Sara\\\1}"
  ]
  N3 -> N4 [
    color = "#2e3436"
    fontcolor = "#2e3436"
    label = "FRIEND\n"
  ]
  N4 [
    label = "{Person|name = \\'Maria\\\1}"
  ]
}
```

Now follows the table of intermediate results and the state of the graph after each clause for the following query:

```
MATCH (john:Person {name: 'John'})
MATCH (john)-[:FRIEND]->(friend)
RETURN friend.name AS friendName
```

The query only has read clauses, so the state of the graph remains unchanged and is therefore omitted below.

Table 1. The table of intermediate results after each clause

Clause	Table of intermediate results after the clause
MATCH (john:Person {name: 'John'})	john ({name: 'John'})

Clause	Table of intermediate results after the clause						
MATCH (john)-[:FRIEND]->(friend)	<table border="1"> <tr> <td>john</td><td>friend</td></tr> <tr> <td>{name: 'John'}</td><td>{name: 'Sara'}</td></tr> <tr> <td>{name: 'John'}</td><td>{name: 'Joe'}</td></tr> </table>	john	friend	{name: 'John'}	{name: 'Sara'}	{name: 'John'}	{name: 'Joe'}
john	friend						
{name: 'John'}	{name: 'Sara'}						
{name: 'John'}	{name: 'Joe'}						
RETURN friend.name AS friendName	<table border="1"> <tr> <td>friendName</td></tr> <tr> <td>'Sara'</td></tr> <tr> <td>'Joe'</td></tr> </table>	friendName	'Sara'	'Joe'			
friendName							
'Sara'							
'Joe'							

The above example only looked at clauses that allow linear composition and omitted write clauses. The next section will explore these non-linear composition and write clauses.

Read-write queries

In a Cypher query, read and write clauses can take turns. The most important aspect of read-write queries is that the state of the graph also changes between clauses.



A clause can never observe writes made by a later clause.

Example 7. Table of intermediate results and state of the graph between read and write clauses

Using the same example graph as above, this example shows the table of intermediate results and the state of the graph after each clause for the following query:

```
MATCH (j:Person) WHERE j.name STARTS WITH "J"
CREATE (j)-[:FRIEND]->(jj:Person {name: "Jay-jay"})
```

The query finds all nodes where the `name` property starts with "J" and for each such node it creates another node with the `name` property set to "Jay-jay".

Table 2. The table of intermediate results and the state of the graph after each clause

Clause	Table of intermediate results after the clause	State of the graph after the clause, changes in red			
MATCH (j:Person) WHERE j.name STARTS WITH "J"	<table border="1"> <tr> <td>j</td> </tr> <tr> <td>{name: 'John'}</td> </tr> <tr> <td>{name: 'Joe'}</td> </tr> </table>	j	{name: 'John'}	{name: 'Joe'}	<pre>digraph L { node [shape=record style=rounded]; N0 [label = "{Person name = '\'John\'\l}'"] N0 -> N3 [color = "grey" fontcolor = "grey" label = "FRIEND\n"] N0 -> N1 [color = "#2e3436" fontcolor = "#2e3436" label = "FRIEND\n"] N1 [label = "{Person name = '\'Joe\'\l}'"] N1 -> N2 [color = "grey" fontcolor = "grey" label = "FRIEND\n"] N2 [color = "grey" fontcolor = "grey" label = "{Person name = '\'Steve\'\l}'"] N3 [color = "grey" fontcolor = "grey" label = "{Person name = '\'Sara\'\l}'"] N3 -> N4 [color = "grey" fontcolor = "grey" label = "FRIEND\n"] N4 [color = "grey" fontcolor = "grey" label = "{Person name = '\'Maria\'\l}'"]</pre>
j					
{name: 'John'}					
{name: 'Joe'}					

Clause	Table of intermediate results after the clause		State of the graph after the clause, changes in red
CREATE (j)-[:FRIEND]->(jj:Person {name: "Jay-jay"})	j	jj	<pre> graph LR N0["{Person name = 'John'\l}"] --> N3["{Person name = 'Sara'\l}"] N0["{Person name = 'John'\l}"] --> N1["{Person name = 'Joe'\l}"] N1["{Person name = 'Joe'\l}"] --> N2["{Person name = 'Steve'\l}"] N3["{Person name = 'Sara'\l}"] --> N4["{Person name = 'Maria'\l}"] N0["{Person name = 'John'\l}"] --> N5["{Person name = 'Jay-jay'\l}"] N1["{Person name = 'Joe'\l}"] --> N6["{Person name = 'Jay-jay'\l}"] </pre>

It is important to note that the **MATCH** clause does not find the **Person** nodes that are created by the **CREATE** clause, even though the name "Jay-jay" starts with "J". This is because the **CREATE** clause

comes after the `MATCH` clause and thus the `MATCH` can not observe any changes to the graph made by the `CREATE`.

Queries with `UNION`

`UNION` queries are slightly different because the results of two or more queries are put together, but each query starts with an empty table of intermediate results.

In a query with a `UNION` clause, any clause before the `UNION` cannot observe writes made by a clause after the `UNION`. Any clause after `UNION` can observe all writes made by a clause before the `UNION`. This means that the rule that a clause can never observe writes made by a later clause still applies in queries using `UNION`.

Example 8. Table of intermediate results and state of the graph in a query with UNION

Using the same example graph as above, this example shows the table of intermediate results and the state of the graph after each clause for the following query:

```

CREATE (jj:Person {name: "Jay-jay"})
RETURN count(*) AS count
UNION
MATCH (j:Person) WHERE j.name STARTS WITH "J"
RETURN count(*) AS count
  
```

Table 3. The table of intermediate results and the state of the graph after each clause

Clause	Table of intermediate results after the clause	State of the graph after the clause, changes in red		
CREATE (jj:Person {name: "Jay-jay"})	<table border="1"> <tr> <td>jj</td> </tr> <tr> <td>{name: 'Jay-jay'}</td> </tr> </table>	jj	{name: 'Jay-jay'}	<pre> graph LR N0["{Person name = 'John'\\"1}"] N1["{Person name = 'Joe'\\"1}"] N2["{Person name = 'Steve'\\"1}"] N3["{Person name = 'Sara'\\"1}"] N4["{Person name = 'Maria'\\"1}"] N5["{Person name = 'Jay-jay'\\"1}"] N0 --> N1 N0 --> N2 N0 --> N3 N0 --> N4 N1 --> N2 N1 --> N3 N1 --> N4 N2 --> N3 N2 --> N4 N3 --> N4 N5 --> N4 </pre>
jj				
{name: 'Jay-jay'}				

Clause	Table of intermediate results after the clause	State of the graph after the clause, changes in red		
RETURN count(*) AS count	<table border="1"> <thead> <tr> <th>count</th></tr> </thead> <tbody> <tr> <td>1</td></tr> </tbody> </table>	count	1	<pre> digraph L { node [shape=record style=rounded]; N0 [label = "{Person name = \'John\'\l}"] N0 -> N3 [color = "#2e3436" fontcolor = "#2e3436" label = "FRIEND\n"] N0 -> N1 [color = "#2e3436" fontcolor = "#2e3436" label = "FRIEND\n"] N1 [label = "{Person name = \'Joe\'\l}"] N1 -> N2 [color = "#2e3436" fontcolor = "#2e3436" label = "FRIEND\n"] N2 [label = "{Person name = \'Steve\'\l}"] N3 [label = "{Person name = \'Sara\'\l}"] N3 -> N4 [color = "#2e3436" fontcolor = "#2e3436" label = "FRIEND\n"] N4 [label = "{Person name = \'Maria\'\l}"] N5 [label = "{Person name = \'Jay- jay\'\l}"] } </pre>
count				
1				

Clause	Table of intermediate results after the clause	State of the graph after the clause, changes in red
MATCH (j:Person) WHERE j.name STARTS WITH "J"	j ({name: 'John'}) ({name: 'Joe'}) ({name: 'Jay-jay'})	<pre> digraph L { node [shape=record style=rounded]; N0 [label = "{Person name = '\John'\l}"] N0 -> N3 [color = "grey" fontcolor = "grey" label = "FRIEND\n"] N0 -> N1 [color = "#2e3436" fontcolor = "#2e3436" label = "FRIEND\n"] N1 [label = "{Person name = '\Joe'\l}"] N1 -> N2 [color = "grey" fontcolor = "grey" label = "FRIEND\n"] N2 [color = "grey" fontcolor = "grey" label = "{Person name = '\Steve'\l}"] N3 [color = "grey" fontcolor = "grey" label = "{Person name = '\Sara'\l}"] N3 -> N4 [color = "grey" fontcolor = "grey" label = "FRIEND\n"] N4 [color = "grey" fontcolor = "grey" label = "{Person name = '\Maria'\l}"] N5 [label = "{Person name = '\Jay- jay'\l}"] } </pre>

Clause	Table of intermediate results after the clause	State of the graph after the clause, changes in red		
RETURN count(*) AS count	<table border="1"> <tr> <td>count</td></tr> <tr> <td>3</td></tr> </table>	count	3	<pre> digraph L { node [shape=record style=rounded]; N0 [label = "{Person name = \'John\'\l}"] N0 -> N3 [color = "#2e3436" fontcolor = "#2e3436" label = "FRIEND\n"] N0 -> N1 [color = "#2e3436" fontcolor = "#2e3436" label = "FRIEND\n"] N1 [label = "{Person name = \'Joe\'\l}"] N1 -> N2 [color = "#2e3436" fontcolor = "#2e3436" label = "FRIEND\n"] N2 [label = "{Person name = \'Steve\'\l}"] N3 [label = "{Person name = \'Sara\'\l}"] N3 -> N4 [color = "#2e3436" fontcolor = "#2e3436" label = "FRIEND\n"] N4 [label = "{Person name = \'Maria\'\l}"] N5 [label = "{Person name = \'Jay- jay\'\l}"] } </pre>
count				
3				

It is important to note that the `MATCH` clause finds the `Person` node that is created by the `CREATE` clause. This is because the `CREATE` clause comes before the `MATCH` clause and thus the `MATCH` can observe any changes to the graph made by the `CREATE`.

Queries with `CALL {}` subqueries

Subqueries inside a `CALL {}` clause are evaluated for each incoming input row. This means that write clauses inside a subquery can get executed more than once. The different invocations of the subquery are executed in turn, in the order of the incoming input rows.

Later invocations of the subquery can observe writes made by earlier invocations of the subquery.

Example 9. Table of intermediate results and state of the graph in a query with `CALL {}`

Using the same example graph as above, this example shows the table of intermediate results and the state of the graph after each clause for the following query:

```
MATCH (john:Person {name: 'John'})
SET john.friends = []
WITH john
MATCH (john)-[:FRIEND]->(friend)
WITH john, friend
CALL {
    WITH john, friend
    WITH *, john.friends AS friends
    SET john.friends = friends + friend.name
}
```

Table 4. The table of intermediate results and the state of the graph after each clause

Clause	Table of intermediate results after the clause		State of the graph after the clause, changes in red
MATCH (john:Person {name: 'John'})	john	{name: 'John'}	<pre> digraph L { node [shape=record style=rounded]; N0 [label = "{Person name = \\'John\\'\\l}"] N0 -> N3 [color = "grey" fontcolor = "grey" label = "FRIEND\\n"] N0 -> N1 [color = "grey" fontcolor = "grey" label = "FRIEND\\n"] N1 [color = "grey" fontcolor = "grey" label = "{Person name = \\'Joe\\'\\l}"] N1 -> N2 [color = "grey" fontcolor = "grey" label = "FRIEND\\n"] N2 [color = "grey" fontcolor = "grey" label = "{Person name = \\'Steve\\'\\l}"] N3 [color = "grey" fontcolor = "grey" label = "{Person name = \\'Sara\\'\\l}"] N3 -> N4 [color = "grey" fontcolor = "grey" label = "FRIEND\\n"] N4 [color = "grey" fontcolor = "grey" label = "{Person name = \\'Maria\\'\\l}"] } </pre>

Clause	Table of intermediate results after the clause		State of the graph after the clause, changes in red
SET john.friends = []	john	{name: 'John', friends: []}	<pre> digraph L { node [shape=record style=rounded]; N0 [color = "red" fontcolor = "red" label = "{Person name = 'John'\1 friends = []\1}"] N0 -> N3 [label = "FRIEND\n"] N0 -> N1 [color = "#2e3436" fontcolor = "#2e3436" label = "FRIEND\n"] N1 [label = "{Person name = 'Joe'\1}"] N1 -> N2 [label = "FRIEND\n"] N2 [label = "{Person name = 'Steve'\1}"] N3 [label = "{Person name = 'Sara'\1}"] N3 -> N4 [label = "FRIEND\n"] N4 [label = "{Person name = 'Maria'\1}"] } </pre>

Clause	Table of intermediate results after the clause		State of the graph after the clause, changes in red
MATCH (john)-[:FRIEND]->(friend)	john	friend	<pre> digraph L { node [shape=record style=rounded]; N0 [label = "{Person name = \\'John\\'\l friends = []\l}"] N0 -> N3 [label = "FRIEND\n"] N0 -> N1 [color = "#2e3436" fontcolor = "#2e3436" label = "FRIEND\n"] N1 [label = "{Person name = \\'Joe\\'\l}"] N1 -> N2 [color = "grey" fontcolor = "grey" label = "FRIEND\n"] N2 [color = "grey" fontcolor = "grey" label = "{Person name = \\'Steve\\'\l}"] N3 [label = "{Person name = \\'Sara\\'\l}"] N3 -> N4 [color = "grey" fontcolor = "grey" label = "FRIEND\n"] N4 [color = "grey" fontcolor = "grey" label = "{Person name = \\'Maria\\'\l}"] } </pre>

Clause	Table of intermediate results after the clause			State of the graph after the clause, changes in red
First invocation of WITH *, john.friends AS friends	john	friend	friends	<pre> graph LR N0["{Person name = 'John'\nfriends = []}"] --> N3["{Person name = 'Sara'\nfriends = []}"] N0 --> N1["{Person name = 'Joe'\nfriends = []}"] N1 --> N2["{Person name = 'Steve'\nfriends = []}"] N3 --> N4["{Person name = 'Maria'\nfriends = []}"] </pre>

Clause	Table of intermediate results after the clause			State of the graph after the clause, changes in red
First invocation of SET john.friends = friends + friend.name	john	friend	friends	<pre> graph LR node[shape=record style=rounded;] N0[N0
color = "red"; fontcolor = "red"; label = "{Person name = 'John'\\"1 friends = ['Sara']\\"1}"] N0 --> N3[N3
label = "FRIEND
"] N0 --> N1[N1
color = "#2e3436"; fontcolor = "#2e3436"; label = "FRIEND
"] N1 --> N2[N2
label = "FRIEND
"] N2 --> N3 N3 --> N4[N4
label = "{Person name = 'Maria'\\"1}"] </pre>

Clause	Table of intermediate results after the clause			State of the graph after the clause, changes in red
Second invocation of WITH *, john.friends AS friends	john ({name: 'John', friends: ['Sara']})	friend ({name: 'Joe'})	friends ['Sara']	<pre> digraph L { node [shape=record style=rounded]; N0 [label = "{Person name = \'John\'\l friends = ['Sara']\l}"] N0 -> N3 [label = "FRIEND\n"] N0 -> N1 [color = "#2e3436" fontcolor = "#2e3436" label = "FRIEND\n"] N1 [label = "{Person name = \'Joe\'\l}"] N1 -> N2 [label = "FRIEND\n"] N2 [label = "{Person name = \'Steve\'\l}"] N3 [label = "{Person name = \'Sara\'\l}"] N3 -> N4 [label = "FRIEND\n"] N4 [label = "{Person name = \'Maria\'\l}"] } </pre>

Clause	Table of intermediate results after the clause			State of the graph after the clause, changes in red
Second invocation of SET john.friends = friends + friend.name	john	friend	friends	<pre> graph LR node[shape=record style=rounded;] N0["color = \"red\" fontcolor = \"red\" label = \"{Person name = \'John\'\\1 friends = ['Sara', 'Joe']\\1}\""] N0 --> N3["label = \"FRIEND\\n\""] N0 --> N1["color = \"#2e3436\" fontcolor = \"#2e3436\" label = \"FRIEND\\n\""] N1["label = \"{Person name = \'Joe\'\\1}\""] N1 --> N2["label = \"FRIEND\\n\""] N2["label = \"{Person name = \'Steve\'\\1}\""] N3["label = \"{Person name = \'Sara\'\\1}\""] N3 --> N4["label = \"FRIEND\\n\""] N4["label = \"{Person name = \'Maria\'\\1}\""] </pre>

It is important to note that, in the subquery, the second invocation of the **WITH** clause could observe the writes made by the first invocation of the **SET** clause.

Notes on the implementation

An easy way to implement the semantics outlined above is to fully execute each clause and materialize the table of intermediate results before executing the next clause. This approach would consume a lot of memory for materializing the tables of intermediate results and would generally not perform well.

Instead, Cypher will in general try to interleave the execution of clauses. This is called **lazy evaluation**. It only materializes intermediate results when needed. In many read-write queries it is unproblematic to execute clauses interleaved, but when it is not, Cypher must ensure that the table of intermediate results gets materialized at the right time(s). This is done by inserting an **Eager** operator into the execution plan.

Syntax

- Values and types
- Naming rules and recommendations
- Expressions
 - Expressions in general
 - Note on string literals
 - Note on number literals
 - CASE Expressions
 - Subquery expressions
 - EXISTS subqueries
 - COUNT subqueries
 - Label expressions
 - Relationship type expressions
- Variables
- Reserved keywords
- Parameters
 - String literal
 - Regular expression
 - Case-sensitive string pattern matching
 - Create node with properties
 - Create multiple nodes with properties
 - Setting all properties on a node
 - SKIP and LIMIT
 - Node id
 - Multiple node ids
 - Calling procedures
- Operators
 - Operators at a glance
 - Aggregation operators
 - Property operators
 - Mathematical operators
 - Comparison operators
 - Boolean operators
 - String operators

- Temporal operators
- Map operators
- List operators
- Comments
- Patterns
 - Patterns for nodes
 - Patterns for related nodes
 - Patterns for labels
 - Specifying properties
 - Patterns for relationships
 - Variable-length pattern matching
 - Assigning to path variables
- Temporal (Date/Time) values
 - Time zones
 - Temporal instants
 - Specifying temporal instants
 - Specifying dates
 - Specifying times
 - Specifying time zones
 - Examples
 - Accessing components of temporal instants
 - Durations
 - Specifying durations
 - Examples
 - Accessing components of durations
 - Examples
 - Temporal indexing
- Spatial values
 - Introduction
 - Coordinate Reference Systems
 - Geographic coordinate reference systems
 - Cartesian coordinate reference systems
 - Spatial instants
 - Creating points
 - Accessing components of points

- Point index
- Lists
 - Lists in general
 - List comprehension
 - Pattern comprehension
- Maps
 - Literal maps
 - Map projection
- Working with null
 - Introduction to null in Cypher
 - Logical operations with null
 - The [\] operator and null
 - The IN operator and null
 - Expressions that return null

Values and types

This section provides an overview of data types in Cypher.

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

Property types

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

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

Cypher also provides pass-through support for byte arrays, which can be stored as property values. Byte arrays are 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 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), Id of start node, and Id of end node.

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
- Composite types can be constructed with [Cypher literals](#)

The [List](#) data type is a heterogeneous, ordered collection of values, each of which can have any property, structural or composite type.

As noted above, homogeneous lists of simple types can be stored as properties.

The [Map](#) data type is a heterogeneous, unordered collection of (Key, Value) pairs, where Key is a string 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
Boolean	False	True	-
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	P292471208677Y6M15DT15H36M32.9999999S	Nanoseconds
Float	Double.MIN_VALUE ^[1]	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	Cartesian: (-Double.MAX_VALUE, -Double.MAX_VALUE) Cartesian_3D: (-Double.MAX_VALUE, -Double.MAX_VALUE, -Double.MAX_VALUE) WGS_84: (-180, -90) WGS_84_3D: (-180, -90, -Double.MAX_VALUE)	Cartesian: (Double.MAX_VALUE, Double.MAX_VALUE) Cartesian_3D: (Double.MAX_VALUE, Double.MAX_VALUE, Double.MAX_VALUE) WGS_84: (180, 90) WGS_84_3D: (180, 90, Double.MAX_VALUE)	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	2^63-1
Long.MIN_VALUE	-2^63

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>)

	<p>Using escaped names with unsanitized user input makes you vulnerable to Cypher injection. Some techniques to mitigate this are:</p> <ul style="list-style-type: none">• 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`): `0o1372`, `-0o5671`.
- A string literal: `'Hello'`, `"World"`.
- A float literal: `Inf`, `Infinity`, `NaN`
- 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`.
- A special case of predicates are label and relationship type expressions: `(n:A|B), ()-[r:R1|R2]->()`.
- A subquery expression. For example: `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
<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.

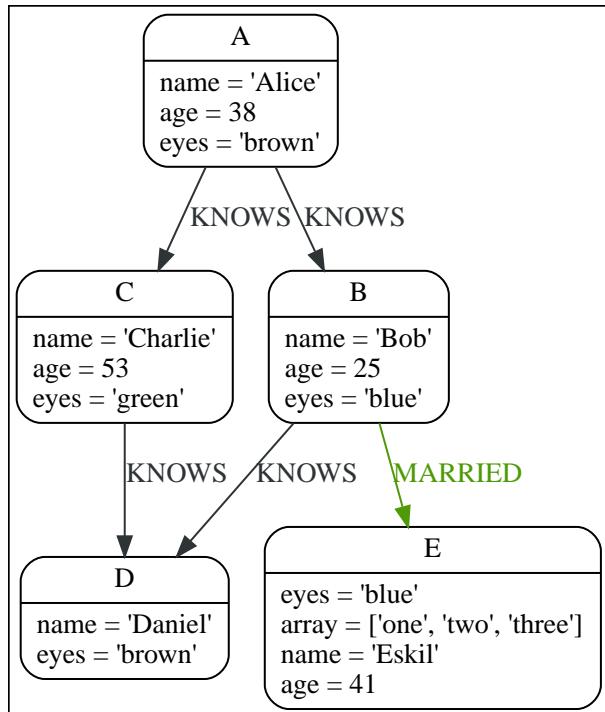
CASE expressions

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
<code>test</code>	A valid expression.
<code>value</code>	An expression whose result will be compared to <code>test</code> .

Name	Description
result	This is the expression returned as output if <code>value</code> matches <code>test</code> .
default	If no match is found, <code>default</code> is returned.

Query

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

Table 5. 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
predicate	A predicate that is tested to find a valid alternative.
result	This is the expression returned as output if <code>predicate</code> evaluates to <code>true</code> .
default	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 6. 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 7. 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 8. 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 9. Result

(empty result)
Rows: 0
Properties set: 5

Using **CASE** with null values

When using the simple **CASE** form, it is useful to remember that in Cypher `null = null` yields `null`.

Example 10. CASE

For example, you might expect `age_10_years_ago` to be `-1` for the node named `Daniel`:

Query

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

However, as `null = null` does not yield `true`, the `WHEN null THEN -1` branch is never taken, resulting in the `ELSE n.age - 10` branch being taken instead, returning `null`.

Table 10. Result

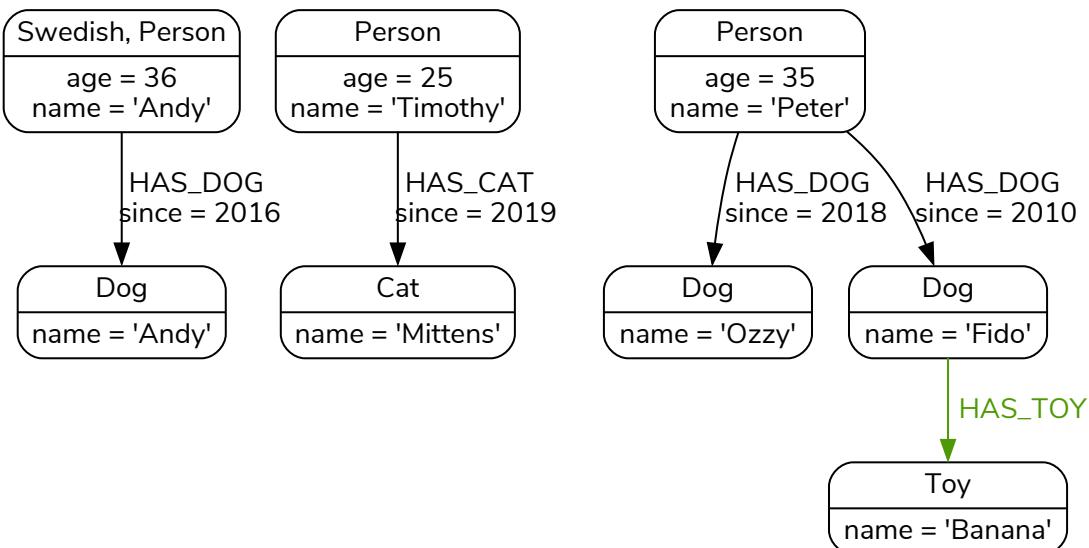
n.name	age_10_years_ago
"Alice"	28
"Bob"	15
"Charlie"	43
"Daniel"	<null>
"Eskil"	31

Rows: 5

Subquery expressions

Subquery expressions can appear anywhere that an expression is valid. 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.

The following graph is used for the examples below:



EXISTS subqueries

An `EXISTS` subquery can be used to find out if a specified pattern exists at least once in the data. It serves the same purpose as a `path pattern` but is more powerful because it allows you to use `MATCH` and `WHERE` clauses internally. Moreover, it can appear in any expression position, unlike path patterns. If the subquery evaluates to at least one row, the whole expression will become `true`. This also means that the system only needs to evaluate if there is at least one row and can skip the rest of the work.

Any non-writing query is allowed. `EXISTS` subqueries differ from regular queries in that the final `RETURN` clause may be omitted, as any variable defined within the subquery will not be available outside of the expression, even if a final `RETURN` clause is used.

It is worth noting that the `MATCH` keyword can be omitted in subqueries in cases where the `EXISTS` consists of only a pattern and an optional `WHERE` clause.

Simple `EXISTS` subquery

Variables introduced by the outside scope can be used in the `EXISTS` subquery without importing them. In this regard, `EXISTS` subqueries are different from `CALL` subqueries, which do require importing. The following example shows this:

Query

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

Table 11. Result

name
"Andy"
"Peter"
Rows: 2

`EXISTS` 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 12. Result

name
"Andy"
Rows: 1

Nesting `EXISTS` subqueries

`EXISTS` 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 from 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 13. Result

name
"Peter"
Rows: 1

`EXISTS` subquery outside of a `WHERE` clause

`EXISTS` subquery expressions can appear anywhere that an expression is valid. Here the result is a boolean that shows whether the subquery can find the given pattern.

Query

```

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

```

Table 14. Result

name	hasDog
"Andy"	true
"Timothy"	false
"Peter"	true
Rows: 3	

EXISTS subquery with a UNION

`Exists` can be used with a `UNION` clause, and the `RETURN` clauses are not required. It is worth noting that if one branch has a `RETURN` clause, then all branches require one. The below example demonstrates that if one of the `UNION` branches was to return at least one row, the entire `EXISTS` expression will evaluate to true.

Query

```
MATCH (person:Person)
RETURN
    person.name AS name,
    EXISTS {
        MATCH (person)-[:HAS_DOG]->(:Dog)
        UNION
        MATCH (person)-[:HAS_CAT]->(:Cat)
    } AS hasPet
```

Table 15. Result

name	hasPet
"Andy"	true
"Timothy"	true
"Peter"	true
Rows: 3	

EXISTS subquery with WITH

Variables from the outside scope are visible for the entire subquery, even when using a `WITH` clause. This means that shadowing of these variables is not allowed. An outside scope variable is shadowed when a newly introduced variable within the inner scope is defined with the same variable. In the below example, a `WITH` clause introduces a new variable. Note that the outer scope variable `person` referenced in the main query is still available after the `WITH` clause.

Query

```
MATCH (person:Person)
WHERE EXISTS {
    WITH "Ozzy" AS dogName
    MATCH (person)-[:HAS_DOG]->(d:Dog)
    WHERE d.name = dogName
}
RETURN person.name AS name
```

Table 16. Result

name
"Peter"
Rows: 1

EXISTS subquery with RETURN

EXISTS subqueries do not require a `RETURN` clause at the end of the subquery. If one is present, it does not need to be aliased, which is different compared to `CALL` subqueries. Any variables returned in an `EXISTS` subquery will not be available after the subquery.

Query

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

Table 17. Result

name
"Andy"
"Peter"
Rows: 2

COUNT subqueries

A `COUNT` subquery expression can be used to count the number of rows returned by the subquery.

Any non-writing query is allowed. `COUNT` subqueries differ from regular queries in that the final `RETURN` clause may be omitted, as any variable defined within the subquery will not be available outside of the expression, even if a final `RETURN` clause is used. One exception to this is that for a `DISTINCT UNION` clause, the `RETURN` clause is still mandatory.

It is worth noting that the `MATCH` keyword can be omitted in subqueries in cases where the `COUNT` consists of only a pattern and an optional `WHERE` clause.

Simple COUNT subquery

Variables introduced by the outside scope can be used in the `COUNT` subquery without importing them. In this regard, `COUNT` subqueries are different from `CALL` subqueries, [which do require importing](#). The following query exemplifies this and outputs the owners of more than one dog:

Query

```
MATCH (person:Person)
WHERE COUNT { (person)-[:HAS_DOG]->(:Dog) } > 1
RETURN person.name AS name
```

Table 18. Result

name
"Peter"

name

| Rows: 1 |

COUNT subquery with WHERE clause

A WHERE clause can be used inside the COUNT pattern. Variables introduced by the MATCH clause and the outside scope can be used in this scope.

Query

```
MATCH (person:Person)
WHERE COUNT {
  (person)-[:HAS_DOG]->(dog:Dog)
  WHERE person.name = dog.name
} = 1
RETURN person.name AS name
```

Table 19. Result

name

| "Andy" |
| Rows: 1 |

COUNT subquery with a UNION

COUNT can be used with a UNION clause. If the UNION clause is distinct, the RETURN clause is required. UNION ALL clauses do not require the RETURN clause. However, it is worth noting that if one branch has a RETURN clause, then all require one. The below example shows the count of pets each person has by using a UNION clause:

Query

```
MATCH (person:Person)
RETURN
  person.name AS name,
  COUNT {
    MATCH (person)-[:HAS_DOG]->(dog:Dog)
    RETURN dog.name AS petName
    UNION
    MATCH (person)-[:HAS_CAT]->(cat:Cat)
    RETURN cat.name AS petName
  } AS numPets
```

Table 20. Result

name	numPets
------	---------

"Andy"	1
"Timothy"	1
"Peter"	2
Rows: 3	

COUNT subquery with WITH

Variables from the outside scope are visible for the entire subquery, even when using a `WITH` clause. This means that shadowing of these variables is not allowed. An outside scope variable is shadowed when a newly introduced variable within the inner scope is defined with the same variable. In the below example, a `WITH` clause introduces a new variable. Note that the outer scope variable `person` referenced in the main query is still available after the `WITH` clause.

Query

```
MATCH (person:Person)
WHERE COUNT {
    WITH "Ozzy" AS dogName
    MATCH (person)-[:HAS_DOG]->(d:Dog)
    WHERE d.name = dogName
} = 1
RETURN person.name AS name
```

Table 21. Result

name
"Peter"
Rows: 1

Using COUNT subqueries inside other clauses

`COUNT` can be used in any position in a query, with the exception of administration commands, where it is restricted. See a few examples below:

Using COUNT in RETURN

Query

```
MATCH (person:Person)
RETURN person.name, COUNT { (person)-[:HAS_DOG]->(:Dog) } as howManyDogs
```

Table 22. Result

person.name	howManyDogs
"Andy"	1
"Timothy"	0
"Peter"	2
Rows: 3	

Using COUNT in SET

Query

```
MATCH (person:Person) WHERE person.name ="Andy"
SET person.howManyDogs = COUNT { (person)-[:HAS_DOG]->(:Dog) }
RETURN person.howManyDogs as howManyDogs
```

Table 23. Result

howManyDogs
1

Rows: 1
Properties set: 1

Using COUNT in CASE

Query

```
MATCH (person:Person)
RETURN
CASE
    WHEN COUNT { (person)-[:HAS_DOG]->(:Dog) } > 1 THEN "Doglover " + person.name
    ELSE person.name
END AS result
```

Table 24. Result

result
"Andy"
"Timothy"
"Doglover Peter"
Rows: 3

Using COUNT as a grouping key

The following query groups all persons by how many dogs they own, and then calculates the average age for each group.

Query

```
MATCH (person:Person)
RETURN COUNT { (person)-[:HAS_DOG]->(:Dog) } AS numDogs,
        avg(person.age) AS averageAge
ORDER BY numDogs
```

Table 25. Result

numDogs	averageAge
0	25.0
1	36.0

numDogs	averageAge
2	35.0
Rows: 3	

COUNT subquery with RETURN

COUNT subqueries do not require a `RETURN` clause at the end of the subquery. If one is present, it does not need to be aliased. This is a difference compared to from `CALL subqueries`. Any variables returned in a `COUNT` subquery will not be available after the subquery.

Query

```

MATCH (person:Person)
WHERE COUNT {
    MATCH (person)-[:HAS_DOG]->(:Dog)
    RETURN person.name
} = 1
RETURN person.name AS name
  
```

Table 26. Result

name
"Andy"
Rows: 1

Label expressions

In earlier versions of Neo4j, label expressions for nodes had a single colon operator that represented the `AND` operator. With the release of version 5, a new label expression with an extended set of logical operators is being introduced, in addition to the single colon operator. It is important to note that you cannot mix these different types of label expression syntax. For more information, see [Restrictions on using the different types of label expression syntax](#).

Label expressions evaluate to `true` or `false` when applied to the set of labels for a node.

Assuming no other filters are applied, then a label expression evaluating to `true` means the node is matched.

The following table displays whether the label expression matches the relationship:

Table 27. Label expression matches

Label expression	Node							
	(<code>)</code>	(<code>:A</code>)	(<code>:B</code>)	(<code>:C</code>)	(<code>:A:B</code>)	(<code>:A:C</code>)	(<code>:B:C</code>)	(<code>:A:B:C</code>)
(<code>)</code>	✓	✓	✓	✓	✓	✓	✓	✓
(<code>:A</code>)		✓			✓	✓		✓
(<code>:A&B</code>)					✓			✓

(:A B)		✓	✓		✓	✓	✓	✓	✓
(:!A)	✓		✓	✓			✓		
(:!!A)		✓			✓	✓			✓
(:A&!A)									
(:A !A)	✓	✓	✓	✓	✓	✓	✓	✓	✓
(:%)		✓	✓	✓	✓	✓	✓	✓	✓
(:!%)	✓								
(:% !%)	✓	✓	✓	✓	✓	✓	✓	✓	✓
(:&%!%)									
(:A&%)		✓			✓	✓			✓
(:A %)		✓	✓	✓	✓	✓	✓	✓	✓
(:(A&B)&!B&C))					✓				
(:(A&%)&%)			✓	✓				✓	

Restrictions on using the different types of label expression syntax

Neo4j version 5 introduces an ampersand operator, which is equivalent to the colon conjunction operator. Mixing the colon conjunction operator with any of the new label expression operators in the same clause will raise a syntax error.

For example, each of the following clauses will raise syntax errors:

- MATCH (n:A|B:C)
- MATCH (n:A:B)-[]-(m:(A&B)|C)
- MATCH (n:A:B)--(m), (n)->(o:(A&B)|C)
- RETURN n:A&B, n:A:B
- MATCH (n:A:B)-[]-(m) WHERE m:(A&B)|C

In earlier versions of Neo4j (version 4.4 and earlier), relationship type expressions only had the pipe operator. As the pipe operator will continue to act as an OR operator, it can continue to be used alongside the new operators.

To make it easier to use the new syntax when extending existing queries, using the different syntax types in separate clauses will be supported.

For example, the following query will not raise a syntax error:

```
MATCH (m:A:B:C)-[]->()
MATCH (n:(A&B)|C)-[]->(m)
RETURN m,n
```

Queries that exclusively use syntax from earlier versions of Neo4j (version 4.4 and earlier) will continue to be supported.

For example, the following will not raise a syntax error:

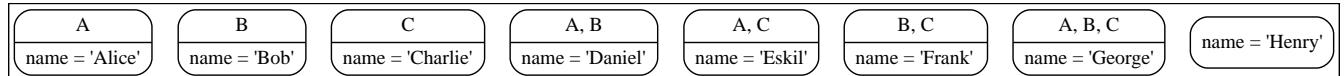
```

MATCH (m:A:B:C)-[:S|T]->()
RETURN
CASE
  WHEN m:D:E THEN m.p
  ELSE null
END AS result

```

Examples

The following graph is used for the examples below:



- [Node pattern without label expressions](#)
- [Node pattern with a single node label](#)
- [Node pattern with an AND expression for the node labels](#)
- [Node pattern with an OR expression for the node labels](#)
- [Node pattern with NOT expressions for the node labels](#)
- [Node pattern with a Wildcard expression for the node labels](#)
- [Node pattern with nested label expressions](#)
- [WHERE clause with label expressions as a predicate](#)
- [Label expressions in the WITH and RETURN clauses](#)

Node pattern without label expressions

A node pattern without a label expression returns all nodes in the graph, including nodes without labels.

Example 11. Label expression

Query

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

Table 28. Result

name
"Alice"
"Bob"
"Charlie"
"Daniel"
"Eskil"
"Frank"
"George"
"Henry"
Rows: 8

Node pattern with a single node label

A node pattern with a single label returns the nodes that contain the specified label.

Example 12. Label expression

Query

```
MATCH (n:A)
RETURN n.name AS name
```

Table 29. Result

name
"Alice"
"Daniel"
"Eskil"
"George"
Rows: 4

Node pattern with an AND expression for the node labels

A node pattern with an AND expression for the node label returns the nodes that contain both of the specified labels.

Example 13. Label expression

Query

```
MATCH (n:A&B)
RETURN n.name AS name
```

Table 30. Result

name
"Daniel"
"George"
Rows: 2

Node pattern with an OR expression for the node labels

A match with OR expressions for the node label returns the nodes that contain either of the specified labels.

Example 14. Label expression

Query

```
MATCH (n:A|B)
RETURN n.name AS name
```

Table 31. Result

name
"Alice"
"Bob"
"Daniel"
"Eskil"
"Frank"
"George"
Rows: 6

Node pattern with NOT expressions for the node labels

A node pattern with a NOT expression for the node label returns the nodes that do not contain the specified label.

Example 15. Label expression

Query

```
MATCH (n:IA)
RETURN n.name AS name
```

Table 32. Result

name
"Bob"
"Charlie"
"Frank"
"Henry"
Rows: 4

Node pattern with a **Wildcard** expression for the node labels

A node pattern with a **Wildcard** expression for the node label returns all the nodes that contain at least one label.

Example 16. Label expression

Query

```
MATCH (n:%)
RETURN n.name AS name
```

Table 33. Result

name
"Alice"
"Bob"
"Charlie"
"Daniel"
"Eskil"
"Frank"
"George"
Rows: 7

Node pattern with nested label expressions

A node pattern with nested label expressions returns the nodes for which the full expression is **true**.

Example 17. Label expression

Query

```
MATCH (n:(A&B)|C)
RETURN n.name AS name
```

Table 34. Result

name
"Charlie"
"Eskil"
"Frank"
"George"
"Henry"
Rows: 5

WHERE clause with label expressions as a predicate

A label expression can also be used as a predicate in the WHERE clause.

Example 18. Label expression

Query

```
MATCH (n)
WHERE n:A|B
RETURN n.name AS name
```

Table 35. Result

name
"Alice"
"Bob"
"Daniel"
"Eskil"
"Frank"
"George"
Rows: 6

Label expressions in the WITH and RETURN clauses

A label expression can also be used in a WITH or a RETURN clause.

Example 19. Label expression

Query

```
MATCH (n)
RETURN n:A&B
```

Table 36. Result

n:A&B
false
false
false
true
false
false
true
false
Rows: 8

Relationship type expressions

Relationship type expressions evaluate to `true` or `false` when applied to the type of a relationship.

Assuming no other filters are applied, then a relationship type expression evaluating to `true` means the relationship is matched.



Relationships must have exactly one type. So for example the expressions: `(a)-[r:R&Q]-(b)` or `(a)-[r:!%]-(b)` will never return any results.



Variable length relationships may only have relationship type expressions consisting of `|`. That means that `()-[r:!R*]-()` is not allowed, whereas `()-[r:Q|R*]-()` is allowed.



Relationships must have exactly one type. For example `(a)-[r:R&Q]-(b)` or `(a)-[r:!%]-(b)` will never return any results.

The following table displays whether the relationship type expression matches the relationship:

	Relationship		
Relationship type expression	<code>[:A]</code>	<code>[:B]</code>	<code>[:C]</code>
<code>[]</code>	✓	✓	✓
<code>[:A]</code>	✓		
<code>[:A&B]</code>			

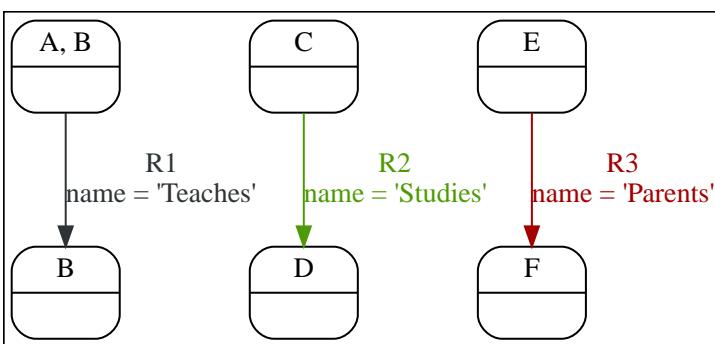
[:A B]	✓	✓	
[:!A]		✓	✓
[:!!A]	✓		
[:A&!A]			
[:A !A]	✓	✓	✓
[:%]	✓	✓	✓
[:!%]			
[:% !%]	✓	✓	✓
[:%&!%]			
[:A&%]			
[:A %]	✓	✓	✓

Label expressions cannot be combined with label syntax. For example, `:A:B&C` will throw an error. Instead, use either `:A&B&C` or `:A:B:C`.

Examples:

- Relationship pattern without relationship type expression
- Relationship pattern with a single relationship type
- Relationship pattern with an **OR** expression for the relationship types
- Relationship pattern with a **NOT** expression for the relationship types
- Relationship pattern with a nested relationship type expression
- **WHERE** clause with a relationship type expression in the predicate
- **WITH** and **RETURN** clauses with a relationship type expression
- **CASE** expression with relationship type and label expressions

The following graph is used for the examples below:



Relationship pattern without relationship type expression

A relationship pattern without a relationship type expression returns all relationships in the graph.

Example 20. Relationship type expressions

Query

```
MATCH ()-[r]->()
RETURN r.name as name
```

Table 37. Result

name
"Teaches"
"Studies"
"Parents"
Rows: 3

Relationship pattern with a single relationship type

A relationship pattern with a single relationship type returns the relationships that contain the specified type.

Example 21. Relationship type expression

Query

```
MATCH ()-[r:R1]->()
RETURN r.name AS name
```

Table 38. Result

name
"Teaches"
Rows: 1

Relationship pattern with an OR expression for the relationship types

A relationship pattern with an OR expression for the relationship type returns all relationships that contain either of the specified types.

Example 22. Relationship type expression

Query

```
MATCH ()-[r:R1|R2]->()
RETURN r.name AS name
```

Table 39. Result

name
"Teaches"
"Studies"
Rows: 2

Relationship pattern with a NOT expression for the relationship types

A relationship pattern with a NOT expression for the relationship type returns all relationships that do not contain the specified type.

Example 23. Relationship type expression

Query

```
MATCH ()-[r:NOT R1]->()
RETURN r.name AS name
```

Table 40. Result

name
"Studies"
"Parents"
Rows: 2

Relationship pattern with a nested relationship type expression

A relationship pattern with a nested relationship type expression returns all relationships for which the full expression is true.

Example 24. Relationship type expression

Query

```
MATCH ()-[r:(!R1&!R2)|R3]->()
RETURN r.name AS name
```

Table 41. Result

name
"Parents"
Rows: 1

WHERE clause with a relationship type expression in the predicate

A relationship type expression can also be used as a predicate in the WHERE clause.

Example 25. Relationship type expression

Query

```
MATCH (n)-[r]->(m)
WHERE r:R1|R2
RETURN r.name AS name
```

Table 42. Result

name
"Teaches"
"Studies"
Rows: 2

WITH and RETURN clauses with a relationship type expression

A relationship type expression can also be used in the WITH or RETURN clauses.

Example 26. Relationship type expression

Query

```
MATCH (n)-[r]->(m)
RETURN r:R1|R2 AS result
```

Table 43. Result

result
true
true
false
Rows: 3

CASE expression with relationship type and label expressions

A relationship type expression and a label expression can also be used in **CASE** expressions.

Example 27. Relationship type expression

Query

```
MATCH (n)-[r]->(m)
RETURN
CASE
  WHEN n:A&B THEN 1
  WHEN r:|R1&|R2 THEN 2
  ELSE -1
END AS result
```

Table 44. Result

result
1
-1
2
Rows: 3

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`
- `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
- COUNT
- ELSE
- END
- EXISTS
- 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
- 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.$param = 'something'` is invalid
- relationship types; so `MATCH (n)-[:$param]-(m)` is invalid
- labels; so `MATCH (n:$param)` is invalid

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.

Auto-parameterization

From version 5 onwards, even when a query is partially parameterized, 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.

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 elementId(n) = $id  
RETURN n.name
```

Multiple node ids

Parameters

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

Query

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

Calling procedures

Parameters

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

Query

```
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>
Comparison operators	<code>=, <>, <, >, <=, >=, IS NULL, IS NOT NULL</code>
String-specific comparison operators	<code>STARTS WITH, ENDS WITH, CONTAINS, =~</code> (regex matching)
Boolean operators	<code>AND, OR, XOR, 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 45. 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 46. Result

p.name
"Jane"
"Tom"

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 47. 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 48. Result

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

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 49. 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 50. 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 51. 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 52. 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 53. 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 [Values and types](#)) by equality using the `=` and `<>` operators.

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

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

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

- Paths are treated as lists of alternating nodes and relationships and are equal to all lists that contain that very same sequence of nodes and relationships.
- Testing any value against `null` with both the `=` and the `<>` operators always evaluates to `null`. This includes `null = null` and `null <> null`. The only way to reliably test if a value `v` is `null` is by using the special `v IS NULL`, or `v IS NOT NULL`, equality operators. `v IS NOT NULL` is equivalent to `NOT(v IS NULL)`.

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

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

Ordering and comparison of values

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

- Numerical values are compared for ordering using numerical order (e.g. `3 < 4` is true).
- All comparability tests (`<`, `<=`, `>`, `>=`) with `java.lang.Double.NaN` evaluate as false. For example, `1 > b` and `1 < b` are both false when `b` is `NaN`.
- String values are compared for ordering using lexicographic order (e.g. `"x" < "xy"`).
- Boolean values are compared for ordering such that `false < true`.
- Comparison of spatial values:
 - Point values can only be compared within the same Coordinate Reference System (CRS) — otherwise, the result will be `null`.
 - For two points `a` and `b` within the same CRS, `a` is considered to be greater than `b` if `a.x > b.x` and `a.y > b.y` (and `a.z > b.z` for 3D points).
 - `a` is considered less than `b` if `a.x < b.x` and `a.y < b.y` (and `a.z < b.z` for 3D points).
 - If none 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 ^[2].

- 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 54. 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 55. 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 56. 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
  localdatetime({year:1984, month:10, day:11, hour:12, minute:31, second:14}) AS aDateTime,
  duration({years: 12, nanoseconds: 2}) AS aDuration
RETURN aDateTime + aDuration, aDateTime - aDuration
```

Table 57. 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 58. 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 59. 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 60. Result

duration1	duration2	duration1 + duration2	duration1 - duration2
P12Y5M14DT16H13M10.0000000 01S	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 61. Result

aDuration	aDuration * 2	aDuration / 3	
P14DT13M10.000000001S	P28DT26M20.000000002S	P4DT16H4M23.33333333S	
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 `\[\]` operator and `null`](#).

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 62. 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 63. Result

result
"Anne"
Rows: 1

More details on maps can be found in [Maps](#).

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 64. 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 65. 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 66. 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 67. 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 68. 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 69. 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 70. Result

result
true
Rows: 1

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

Comments

This section describes how to use comments in Cypher.

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

Examples:

```
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`.

You can specify additional constraints by introducing [node pattern predicates](#).

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.

For more information on how to use relationship type expressions, see [Relationship type expressions](#).

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

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

It is not possible to use the same name for a relationship multiple times within a pattern due to [relationship isomorphism](#).

Example 28. Relationship isomorphism

Using the same variable name for relationships multiple times within a pattern is not allowed.

The following example is therefore not allowed.

```
()-[r:REL_TYPE]-()-[r:REL_TYPE]-()
```

You can specify additional constraints by introducing a [relationship pattern predicate](#).

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)
```

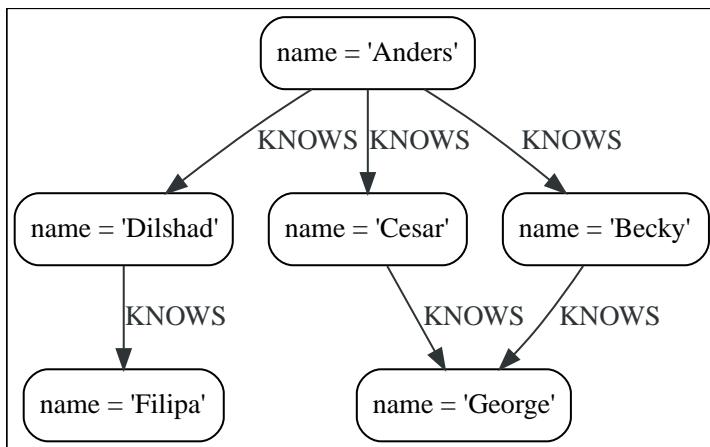
To describe paths of length 5 or less, use:

(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 71. 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`.

Under certain circumstances variable length relationships can be planned with an optimisation, see [VarLength Expand Pruning](#) query plan.

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.

Temporal (Date/Time) values

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



- Refer to [Temporal functions - instant types](#) for information regarding temporal functions allowing for the creation and manipulation of temporal values.
- Refer to [Temporal operators](#) for information regarding temporal operators.
- Refer to [Ordering and comparison of values](#) for information regarding the comparison and ordering of temporal values.

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

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><date></code>
<code>Time</code>	<code><time><timezone></code> or <code>T<time><timezone></code>
<code>LocalTime</code>	<code><time></code> or <code>T<time></code>
<code>DateTime</code> *	<code><date>T<time><timezone></code>
<code>LocalDateTime</code> *	<code><date>T<time></code>

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

Specifying dates

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

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

- 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

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

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

Specifying times

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

Cypher does not support leap seconds; [UTC-SLS](#) (UTC with Smoothed Leap Seconds) is used to manage the difference in time between UTC and TAI (International Atomic Time).

The following formats are supported for specifying times:

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

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

Specifying time zones

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

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

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

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

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

The following formats are supported for specifying time zones:

Format	Description	Example	Supported for DateTime	Supported for Time
Z	UTC	Z	✓	✓
±HH:MM	Hour:Minute	+09:30	✓	✓

Format	Description	Example	Supported for DateTime	Supported for Time
<code>±HH:MM[ZoneName]</code>	Hour:Minute[ZoneName]	<code>+08:45[Australia/Eucla]</code>	✓	
<code>±HHMM</code>	Hour:Minute	<code>+0100</code>	✓	✓
<code>±HHMM[ZoneName]</code>	Hour:Minute[ZoneName]	<code>+0200[Africa/Johannesburg]</code>	✓	
<code>±HH</code>	Hour	<code>-08</code>	✓	✓
<code>±HH[ZoneName]</code>	Hour[ZoneName]	<code>+08[Asia/Singapore]</code>	✓	
<code>[ZoneName]</code>	[ZoneName]	<code>[America/Regina]</code>	✓	

Examples

Here are examples of parsing temporal instant values using various formats.

For more details, refer to [An overview of temporal instant type creation](#).

Example 29. datetime

Parsing a DateTime using the calendar date format:

Query

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

Table 72. Result

theDateTime
2015-06-24T12:50:35.556+01:00
Rows: 1

Example 30. localdatetime

Parsing a LocalDateTime using the ordinal date format:

Query

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

Table 73. Result

theLocalDateTime
2015-07-04T19:32:24
Rows: 1

Example 31. date

Parsing a Date using the week date format:

Query

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

Table 74. Result

theDate
2015-03-26
Rows: 1

Example 32. time

Parsing a Time:

Query

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

Table 75. Result

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

Example 33. localtime

Parsing a LocalTime:

Query

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

Table 76. Result

theLocalTime
12:50:35.556
Rows: 1

Accessing components of temporal instants

Components of temporal instant values can be accessed as properties.

Table 77. Components of temporal instant values and where they are supported

Component	Description	Type	Range/Format	Date	DateTime	LocalDateTime	Time	LocalTime
instant.year	The year component represents the astronomical year number of the instant. ^[3]	Integer	At least 4 digits. For more information, see the rules for using the Year component .	✓	✓	✓		
instant.quarter	The quarter-of-the-year component.	Integer	1 to 4.	✓	✓	✓		
instant.month	The month-of-the-year component.	Integer	1 to 12.	✓	✓	✓		
instant.week	The week-of-the-year component. ^[4]	Integer	1 to 53.	✓	✓	✓		
instant.weekYear	The year that the week-of-year component belongs to. ^[5]	Integer	At least 4 digits. For more information, see the rules for using the Year component .	✓	✓	✓		
instant.dayOfQuarter	The day-of-the-quarter component.	Integer	1 to 92.	✓	✓	✓		
instant.quarterDay	The day-of-the-quarter component (alias for instant.dayOfQuarter).	Integer	1 to 92.	✓	✓	✓		
instant.day	The day-of-the-month component.	Integer	1 to 31.	✓	✓	✓		
instant.ordinalDay	The day-of-the-year component.	Integer	1 to 366.	✓	✓	✓		
instant.dayOfWeek	The day-of-the-week component (the first day of the week is Monday).	Integer	1 to 7.	✓	✓	✓		
instant.weekDay	The day-of-the-week component (alias for instant.dayOfWeek).	Integer	1 to 7.	✓	✓	✓		

Component	Description	Type	Range/Format	Date	DateTime	LocalDateTime	Time	LocalTime
instant.hour	The hour component.	Integer	0 to 23.		✓	✓	✓	✓
instant.minute	The minute component.	Integer	0 to 59.		✓	✓	✓	✓
instant.second	The second component. ^[6]	Integer	0 to 59.		✓	✓	✓	✓
instant.millisecond	The millisecond component.	Integer	0 to 999.		✓	✓	✓	✓
instant.microsecond	The microsecond component.	Integer	0 to 999999.		✓	✓	✓	✓
instant.nanosecond	The nanosecond component.	Integer	0 to 999999999.		✓	✓	✓	✓
instant.timezone	The timezone component.	String	Depending on how the time zone was specified , this is either a time zone name or an offset from UTC in the format <code>±HHMM</code> .		✓		✓	
instant.offset	The timezone offset.	String	In the format <code>±HHMM</code> .		✓		✓	
instant.offsetMinutes	The timezone offset in minutes.	Integer	-1080 to +1080.		✓		✓	
instant.offsetSeconds	The timezone offset in seconds.	Integer	-64800 to +64800.		✓		✓	
instant.epochMillis	The number of milliseconds between <code>1970-01-01T00:00:00+0000</code> and the instant. ^[7]	Integer	Positive for instants after and negative for instants before <code>1970-01-01T00:00:00+0000</code> .		✓			
instant.epochSeconds	The number of seconds between <code>1970-01-01T00:00:00+0000</code> and the instant. ^[8]	Integer	Positive for instants after and negative for instants before <code>1970-01-01T00:00:00+0000</code> .		✓			

Example 34. 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 78. 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 35. 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 79. 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 36. 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 80. Result

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

Example 37. 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 81. Result

d.timezone	d.offset	d.offsetMinutes	d.epochSeconds	d.epochMillis
"Europe/Stockholm"	"+01:00"	60	469020674	469020674645
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
<code>Y</code>	Years	
<code>M</code>	Months	Must be specified before <code>T</code> .
<code>W</code>	Weeks	
<code>D</code>	Days	
<code>H</code>	Hours	
<code>M</code>	Minutes	Must be specified after <code>T</code> .
<code>S</code>	Seconds	

Examples

The following examples demonstrate various methods of parsing Duration values.

For more details, refer to [Creating a Duration from a string](#).

Example 38. duration

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

Query

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

Table 82. Result

theDuration
P14DT16H12M
Rows: 1

Example 39. duration

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

Query

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

Table 83. Result

theDuration
P5M1DT12H
Rows: 1

Example 40. duration

Return a Duration of 45 seconds:

Query

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

Table 84. Result

theDuration
PT45S
Rows: 1

Example 41. duration

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

Query

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

Table 85. Result

theDuration
P17DT12H
Rows: 1

Accessing components of durations

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

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

Component Group	Component	Description	Type	Details
Months	duration.years	The total number of years.	Integer	Each set of 4 quarters is counted as 1 year; each set of 12 months is counted as 1 year.
	duration.quarters	The total number of quarters.	Integer	Each year is counted as 4 quarters; each set of 3 months is counted as 1 quarter.
	duration.months	The total number of months.	Integer	Each year is counted as 12 months; each_quarter_is counted as 3 months.
Days	duration.weeks	The total number of weeks.	Integer	Each set of 7 days is counted as 1 week.
	duration.days	The total number of days.	Integer	Each week is counted as 7 days.

Component Group	Component	Description	Type	Details
Seconds	duration.hours	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.
	duration.minutes	The total number of minutes.	Integer	Each hour is counted as 60 minutes; each set of 60 seconds is counted as 1 minute.
	duration.seconds	The total number of seconds.	Integer	Each hour is counted as 3600 seconds; each minute is counted as 60 seconds.
	duration.milliseconds	The total number of milliseconds	Integer	Each set of 1000 milliseconds is counted as 1 second.
	duration.microseconds	The total number of microseconds.	Integer	Each millisecond is counted as 1000 microseconds.
	duration.nanoseconds	The total number of nanoseconds.	Integer	Each microsecond is counted as 1000 nanoseconds.

	<p>Please note that:</p> <ul style="list-style-type: none"> • 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
duration.quartersOfYear	Months	The number of quarters in the group that do not make a whole year.	Integer
duration.monthsOfYear	Months	The number of months in the group that do not make a whole year.	Integer
duration.monthsOfQuarter	Months	The number of months in the group that do not make a whole quarter.	Integer
duration.daysOfWeek	Days	The number of days in the group that do not make a whole week.	Integer
duration.minutesOfHour	Seconds	The number of minutes in the group that do not make a whole hour.	Integer
duration.secondsOfMinute	Seconds	The number of seconds in the group that do not make a whole minute.	Integer

Component	Component Group	Description	Type
duration.millisecondsOfSecond	Seconds	The number of milliseconds in the group that do not make a whole second.	Integer
duration.microsecondsOfSecond	Seconds	The number of microseconds in the group that do not make a whole second.	Integer
duration.nanosecondsOfSecond	Seconds	The number of nanoseconds in the group that do not make a whole second	Integer

Example 42. 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 86. Result

d.years	d.quarters	d.quartersOfYear	d.months	d.monthsOfYear	d.monthsOfQuarter
1	5	1	17	5	2

Rows: 1

Example 43. 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 87. Result

d.weeks	d.days	d.daysOfWeek
3	25	4

Rows: 1

Example 44. 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 88. Result

d.hours	d.minutes	d.seconds	d.milliseconds	d.microseconds	d.nanoseconds
1	61	3661	3661111	366111111111	3661111111111111
Rows: 1					

Example 45. 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 89. Result

d.minutesOfHour	d.secondsOfMinute	d.millisecondsOfSecond	d.microsecondsOfSecond	d.nanosecondsOfSecond
1	1	111	1111111	111111111111
Rows: 1				

Examples

The following examples illustrate the use of some of the temporal functions and operators.

Refer to [Temporal functions - instant types](#) and [Temporal operators](#) for more details.

Example 46. duration

Create a Duration representing 1.5 days:

Query

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

Table 90. Result

theDuration
P1DT12H
Rows: 1

Example 47. duration.between

Compute the Duration between two temporal instants:

Query

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

Table 91. Result

theDuration
P30Y8M13D
Rows: 1

Example 48. 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 92. Result

theDuration
P299D
Rows: 1

Example 49. date.truncate

Get the first Date of the current year:

Query

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

Table 93. Result

day
2022-01-01
Rows: 1

Example 50. 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 94. Result

thursday
2019-10-03
Rows: 1

Example 51. 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 95. Result

lastDay
2022-07-31
Rows: 1

Example 52. time

Add a Duration to a Date:

Query

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

Table 96. Result

theTime
01:42:19Z
Rows: 1

Example 53. duration

Add two Duration values:

Query

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

Table 97. Result

theDuration
P1M2DT25H
Rows: 1

Example 54. duration

Multiply a Duration by a number:

Query

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

Table 98. Result

theDuration
PT74H54M
Rows: 1

Example 55. duration

Divide a Duration by a number:

Query

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

Table 99. Result

theDuration
PT1H38M
Rows: 1

Example 56. 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 100. Result

lessThanOneDayApart
true
Rows: 1

Example 57. 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 101. 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), and the underlying database supports storing these point values as properties on nodes and relationships.



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

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

Introduction

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

- Each point can have either 2 or 3 dimensions. This means it contains either 2 or 3 64-bit floating point values, which together are called the Coordinate.
- Each point will also be associated with a specific [Coordinate Reference System](#) (CRS) that determines the meaning of the values in the Coordinate.
- Instances of Point and lists of Point can be assigned to node and relationship properties.
- Nodes with Point or List(Point) properties can be indexed using a point index. This is true for all CRS (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:

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

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

Geographic coordinate reference systems

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

- [WGS 84 2D](#)

- A 2D geographic point in the WGS 84 CRS is specified in one of two ways:
 - `longitude` and `latitude` (if these are specified, and the `crs` is not, then the `crs` is assumed to be `WGS-84`)
 - `x` and `y` (in this case the `crs` must be specified, or will be assumed to be `Cartesian`)
- Specifying this CRS can be done using either the name 'wgs-84' or the SRID 4326 as described in [Point\(WGS-84\)](#)

- [WGS 84 3D](#)

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

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

Query

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

Table 102. 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 and can mean anything the user intends them to mean. This also means that when two cartesian points are passed to the `distance` function, the resulting value will be in the same units as the original coordinates. This is true for both 2D and 3D points, as the pythagoras equation used is generalized to any number of dimensions. However, just as you cannot compare geographic points to cartesian points, you cannot calculate the distance between a 2D point and a 3D point. If you need to do that, explicitly transform the one type into the other. For example:

Query

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

Table 103. Result

bad	good
<null>	5.0
Rows: 1	

Spatial instants

Creating points

All point types are created from two components:

- The Coordinate containing either 2 or 3 floating point values (64-bit)
- The Coordinate Reference System (or CRS) defining the meaning (and possibly units) of the values in the Coordinate

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

- Choice of keys:

- If the coordinate is specified using the keys `latitude` and `longitude` the CRS will be assumed to be Geographic and therefore either [WGS-84](#) or [WGS-84-3D](#).
- If instead `x` and `y` are used, then the default CRS would be [Cartesian](#) or [Cartesian-3D](#)

- Number of dimensions:

- If there are 2 dimensions in the coordinate, `x` & `y` or `longitude` & `latitude` the CRS will be a 2D CRS
- If there is a third dimension in the coordinate, `z` or `height` the CRS will be a 3D CRS

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

Query

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

Table 104. Result

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

For the geographic coordinates, it is important to note that the `latitude` value should always lie in the interval `[-90, 90]` and any other value outside this range will throw an exception. The `longitude` value should always lie in the interval `[-180, 180]` and any other value outside this range will be wrapped around to fit in this range. The `height` value and any cartesian coordinates are not explicitly restricted, and any value within the allowed range of the signed 64-bit floating point type will be accepted.

Accessing components of points

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

Table 105. Components of point instances and where they are supported

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

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 106. Result

x	y	crs	srid
3.0	4.0	"cartesian"	7203

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 107. 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

Point index

If there is a [index](#) on a particular `:Label(property)` combination, and a spatial point is assigned to that property on a node with that label, the node will be indexed in a point index.

For point indexing, Neo4j uses space filling curves in 2D or 3D over an underlying generalized B+Tree. Points will be stored in up to four different trees, one for each of the [four coordinate reference systems](#). This allows for both [equality](#) and [range](#) queries using exactly the same syntax and behaviour as for other property types. If two range predicates are used, which define minimum and maximum points, this will effectively result in a [bounding box query](#). In addition, queries using the [distance](#) function can, under the right conditions, also use the index, as described in the section '[Spatial distance searches](#)'.

Comparability and orderability

This means that queries that rely on the comparison of two points using the inequality operators, `<`, `<=`, `>`, and `>=`, or the specific order of an `ORDER BY n.point` query will need to be rewritten.

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

Lists

Cypher has comprehensive support for 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 108. Result

list
[0,1,2,3,4,5,6,7,8,9]
Rows: 1

In the examples, you use the `range` function. It gives you a list containing all numbers between given start and end numbers. Range is inclusive in both ends.

To access individual elements in the list, you can use the square brackets again. This extracts from the start index and up to, but not including, the end index.

Query

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

Table 109. Result

range(0, 10)[3]
3
Rows: 1

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

Query

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

Table 110. Result

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

8

Rows: 1

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

Query

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

Table 111. Result

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

[0,1,2]

Rows: 1

Query

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

Table 112. Result

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

[0,1,2,3,4,5]

Rows: 1

Query

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

Table 113. Result

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

[6,7,8,9,10]

Rows: 1

Query

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

Table 114. Result

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

[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]
```

Table 115. Result

range(0, 10)[15]

<null>

Rows: 1

Query

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

Table 116. Result

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

[5, 6, 7, 8, 9, 10]

Rows: 1

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

Query

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

Table 117. Result

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

3

Rows: 1

List comprehension

List comprehension is a syntactic construct available in Cypher for creating a list based on existing lists. It follows the form of the mathematical set-builder notation (set comprehension) instead of the use of map and filter functions.

Query

```
RETURN [x IN range(0,10) WHERE x % 2 = 0 | x^3] AS result
```

Table 118. Result

result

[0.0, 8.0, 64.0, 216.0, 512.0, 1000.0]

```
result
```

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 119. Result

```
result
```

[0,2,4,6,8,10]

Rows: 1

Query

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

Table 120. Result

```
result
```

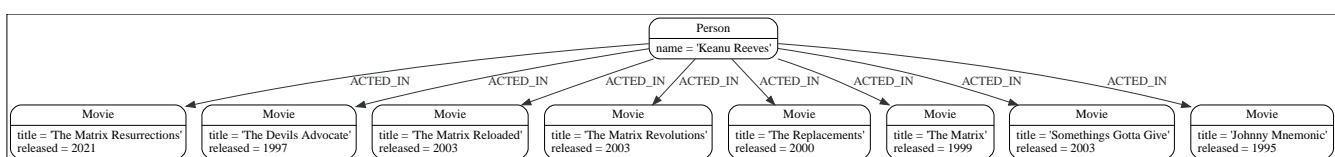
[0.0,1.0,8.0,27.0,64.0,125.0,216.0,343.0,512.0,729.0,1000.0]

Rows: 1

Pattern comprehension

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

The following graph is used for the pattern comprehension examples:



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

Query

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

Table 121. Result

years

[2021, 2003, 2003, 1999]

Rows: 1

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

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

Query

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

Table 122. Result

sorted_years

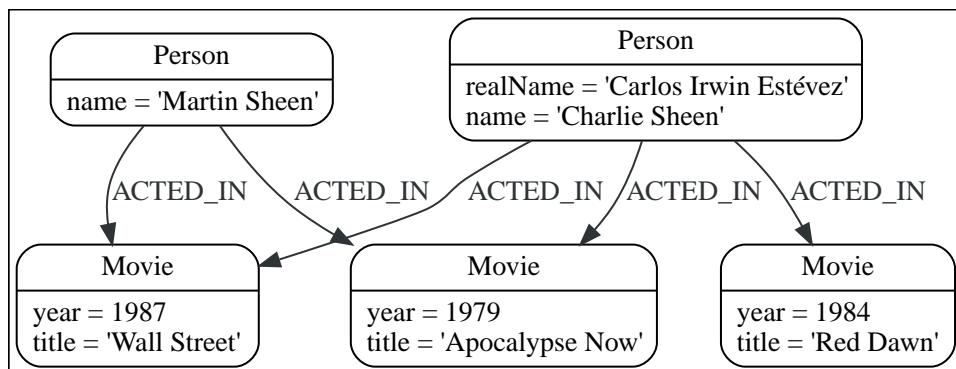
[1995, 1997, 1999, 2000, 2003, 2003, 2003, 2021]

Rows: 1

Maps

This section describes how to use maps in Cyphers.

The following graph is used for the examples below:



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

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

Query

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

Table 123. Result

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

Rows: 1

Map projection

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

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

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

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

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

The following conditions apply:

- If the `map_variable` points to a `null` value, the whole map projection will evaluate to `null`.
- The key names in a map must be of type `String`.

Examples of map projections

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

Query

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

Table 124. Result

actor

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

Rows: 1

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

Query

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

Table 125. Result

actor

```
{nbrOfMovies -> 2, name -> "Martin Sheen"}
{nbrOfMovies -> 3, name -> "Charlie Sheen"}
```

Rows: 2

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

Query

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

Table 126. Result

actor

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

Rows: 1

Working with `null`

This section describes working with the `null` value.

Introduction to `null` in Cypher

In Cypher, `null` is used to represent missing or undefined values. Conceptually, `null` means a missing unknown value and it is treated somewhat differently from other values. For example getting a property from a node that does not have said property produces `null`. Most expressions that take `null` as input will produce `null`. This includes boolean expressions that are used as predicates in the `WHERE` clause. In this case, anything that is not `true` is interpreted as being false.

`null` is not equal to `null`. Not knowing two values does not imply that they are the same value. So the expression `null = null` yields `null` and not `true`.

Logical operations with `null`

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

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

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

The `IN` operator and `null`

The `IN` operator follows similar logic. If Cypher knows that something exists in a list, the result will be `true`. Any list that contains a `null` and doesn't have a matching element will return `null`. Otherwise, the result will be `false`. Here is a table with examples:

Expression	Result
<code>2 IN [1, 2, 3]</code>	<code>true</code>
<code>2 IN [1, null, 3]</code>	<code>null</code>
<code>2 IN [1, 2, null]</code>	<code>true</code>
<code>2 IN [1]</code>	<code>false</code>
<code>2 IN []</code>	<code>false</code>
<code>null IN [1, 2, 3]</code>	<code>null</code>
<code>null IN [1, null, 3]</code>	<code>null</code>
<code>null IN []</code>	<code>false</code>

Using `all`, `any`, `none`, and `single` follows a similar rule. If the result can be calculated definitely, `true` or `false` is returned. Otherwise `null` is produced.

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

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` instead (see [Equality and comparison of values](#)).

[1] 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`.

[2] The `365.2425` days per year comes from the frequency of leap years. A leap year occurs on a year with an ordinal number divisible by `4`, that is not divisible by `100`, unless it is divisible by `400`. This means that over `400` years there are $((365 * 4 + 1) * 25 - 1) * 4 + 1 = 146097$ days, which means an average of `365.2425` days per year.

[3] 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.

[4] The [first week of any year](#) is the week that contains the first Thursday of the year, and thus always contains January 4.

[5] 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.

[6] 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).

[7] The expression `datetime().epochMillis` returns the equivalent value of the `timestamp()` function.

[8] For the nanosecond part of the epoch offset, the regular nanosecond component (`instant.nanosecond`) can be used.

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 be found in [Database management](#) and [Access control](#).

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

Importing data

Clause	Description
<code>LOAD CSV</code>	Use when importing data from CSV files.
<code>CALL { ... } IN TRANSACTIONS</code>	This clause may be used to prevent an out-of-memory error from occurring when importing large amounts of data using <code>LOAD CSV</code> .

Listing functions and procedures

Clause	Description
<code>SHOW FUNCTIONS</code>	List the available functions.
<code>SHOW PROCEDURES</code>	List the available procedures.

Multiple graphs

Clause	Description
<code>USE</code>	Determines which graph a query, or query part, is executed against. Fabric

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><<RETURN, RETURN ... [AS]>></code>	Defines what to include in the query result set.
<code><<WITH, WITH ... [AS]>></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><<UNWIND, UNWIND ... [AS]>></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 <code>MATCH</code> or <code>OPTIONAL MATCH</code> clause or filters the results of a <code>WITH</code> clause.
<code><<ORDERBY, ORDER BY [ASC[ENDING]>> DESC[ENDING]]></code>	A sub-clause following <code>RETURN</code> or <code>WITH</code> , specifying that the output should be sorted in either ascending (the default) or descending order.
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
<code>MERGE</code>	Ensures that a pattern exists in the graph. Either the pattern already exists, or it needs to be created.
<code>--- ON CREATE</code>	Used in conjunction with <code>MERGE</code> , this write sub-clause specifies the actions to take if the pattern needs to be created.
<code>--- ON MATCH</code>	Used in conjunction with <code>MERGE</code> , this write sub-clause specifies the actions to take if the pattern already exists.
<code><<CALLprocedure, CALL ... [YIELD ... >>]</code>	Invokes a procedure deployed in the database and return any results.

Set operations

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

Subquery clauses

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

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

Transaction Commands

Clause	Description
SHOW TRANSACTIONS	List the available transactions.
TERMINATE TRANSACTIONS	Terminate transactions by their IDs.

Writing clauses

These comprise clauses that write the data to the database.

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

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
- Match with a label expression for the node 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 elementId
 - Node by elementId
 - Relationship by elementId
 - Multiple nodes by elementId

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.

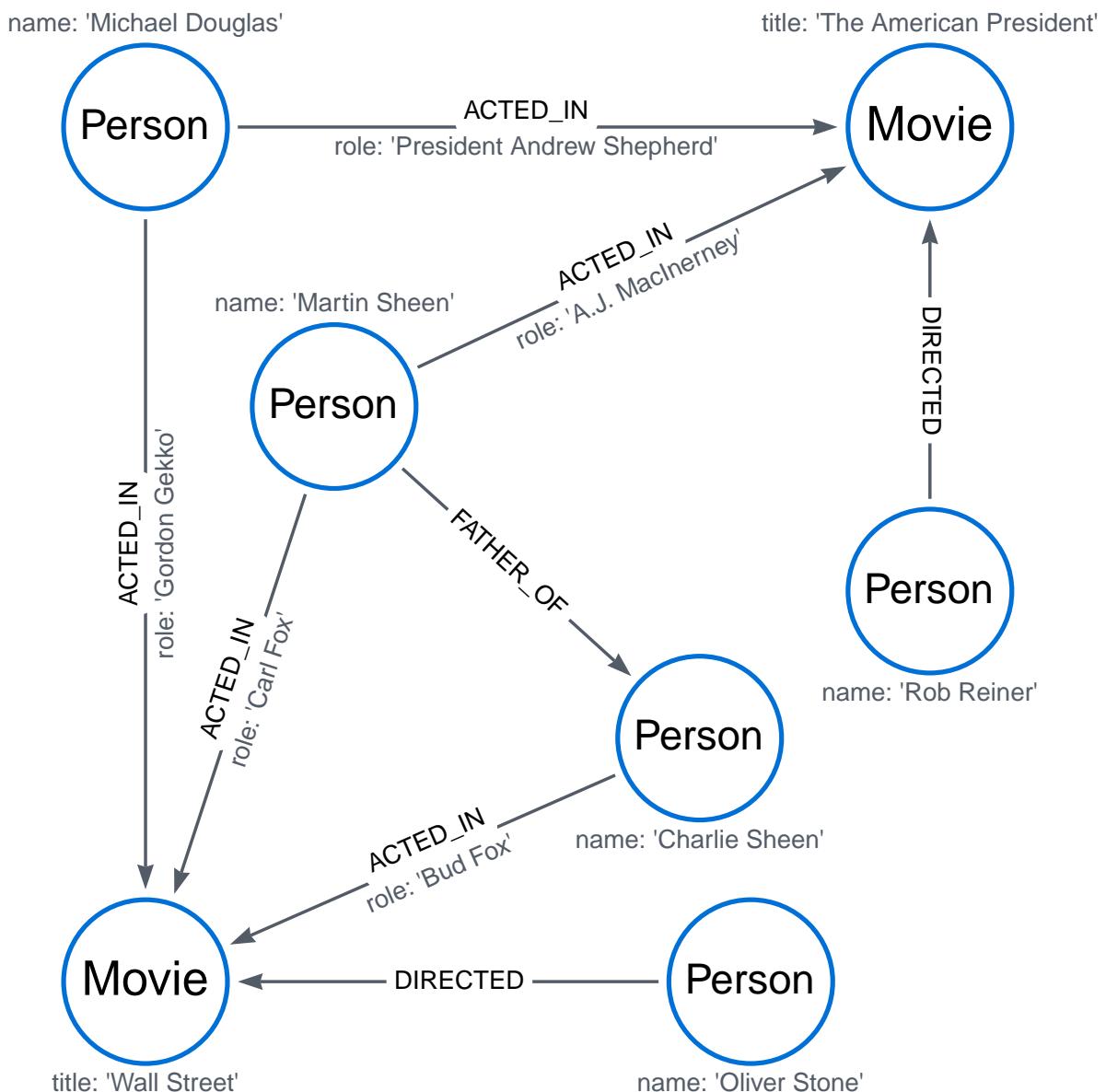
`MATCH` can occur at the beginning of the query or later, possibly after a `WITH`. If it is the first clause, nothing will have been bound yet, and Neo4j will design a search to find the results matching the clause and any associated predicates specified in any `WHERE` part. This could involve a scan 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 the chapter on [Patterns](#).

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 {role: 'Bud Fox'}]->(wallStreet),
(martin)-[:ACTED_IN {role: 'Carl Fox'}]->(wallStreet),
(michael)-[:ACTED_IN {role: 'Gordon Gekko'}]->(wallStreet),
(oliver)-[:DIRECTED]->(wallStreet),
(thePresident:Movie {title: 'The American President'}),
(martin)-[:ACTED_IN {role: 'A.J. MacInerney'}]->(thePresident),
(michael)-[:ACTED_IN {role: 'President Andrew Shepherd'}]->(thePresident),
(rob)-[:DIRECTED]->(thePresident),
(martin)-[:FATHER_OF]->(charlie)
```

Basic node finding

Get all nodes

By 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 127. Result

n
{"name": "Charlie Sheen"}
{"name": "Martin Sheen"}
{"name": "Michael Douglas"}
{"name": "Oliver Stone"}
{"name": "Rob Reiner"}
{"title": "Wall Street"}
{"title": "The American President"}
Rows: 7

Get all nodes with a label

Find all nodes with a specific label:

Query

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

Returns all the nodes with the `Movie` label in the database.

Table 128. 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 129. Result

movie.title
"Wall Street"
Rows: 1

Match with labels

To constrain a pattern with labels on nodes, add the labels to the nodes in the pattern.

Query

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

Returns any nodes with the `Movie` label connected to `Oliver Stone`.

Table 130. Result

movie.title
"Wall Street"
Rows: 1

Match with a label expression for the node labels

A match with an `OR` expression for the node label returns the nodes that contains both the specified labels.

Query

```
MATCH (n:Movie|Person)
RETURN n.name AS name, n.title AS title
```

Table 131. Result

name	title
"Charlie Sheen"	<null>
"Martin Sheen"	<null>
"Michael Douglas"	<null>
"Oliver Stone"	<null>
"Rob Reiner"	<null>
<null>	"Wall Street"
<null>	"The American President"

Rows: 7

Relationship basics

Outgoing relationships

When the direction of a relationship is of interest, it is shown by using `-->` or `<-->`. For example:

Query

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

Returns any nodes connected by an outgoing relationship to the `Person` node with the `name` property set to `Oliver Stone`.

Table 132. Result

movie.title
"Wall Street"
Rows: 1

Directed relationships and variable

It is possible to introduce a variable to a pattern, either for filtering on relationship properties or to return a relationship.

For example: .Query

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

Returns the type of each outgoing relationship from **Oliver Stone**.

Table 133. Result

type(r)
"DIRECTED"
Rows: 1

Match on relationship type

When the relationship type to match on is known, it is possible to specify it by using a colon (:) before the relationship type.

Query

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

Returns all actors who **ACTED_IN** the movie **Wall Street**.

Table 134. Result

actor.name
"Michael Douglas"
"Martin Sheen"
"Charlie Sheen"
Rows: 3

Read more about [relationship type expressions](#).

Match on multiple relationship types

It is possible to match on multiple relationship types by using the pipe symbol (|). For example:

Query

```
MATCH (wallstreet {title: 'Wall Street'})<-[ACTED_IN|DIRECTED]->(person)
RETURN person.name
```

Returns nodes with an **ACTED_IN** or **DIRECTED** relationship to the movie **Wall Street**. .Result

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

Match on relationship type and use a variable

Variables and specific relationship types can be included in the same pattern. For example:

Query

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

Returns the ACTED_IN roles for the movie Wall Street.

Table 135. Result

r.role
"Gordon Gekko"
"Carl Fox"
"Bud Fox"
Rows: 3

Relationships in depth



Relationships will only be matched once inside a single pattern. Read more about this in the section on [uniqueness](#).

Relationship types with uncommon characters

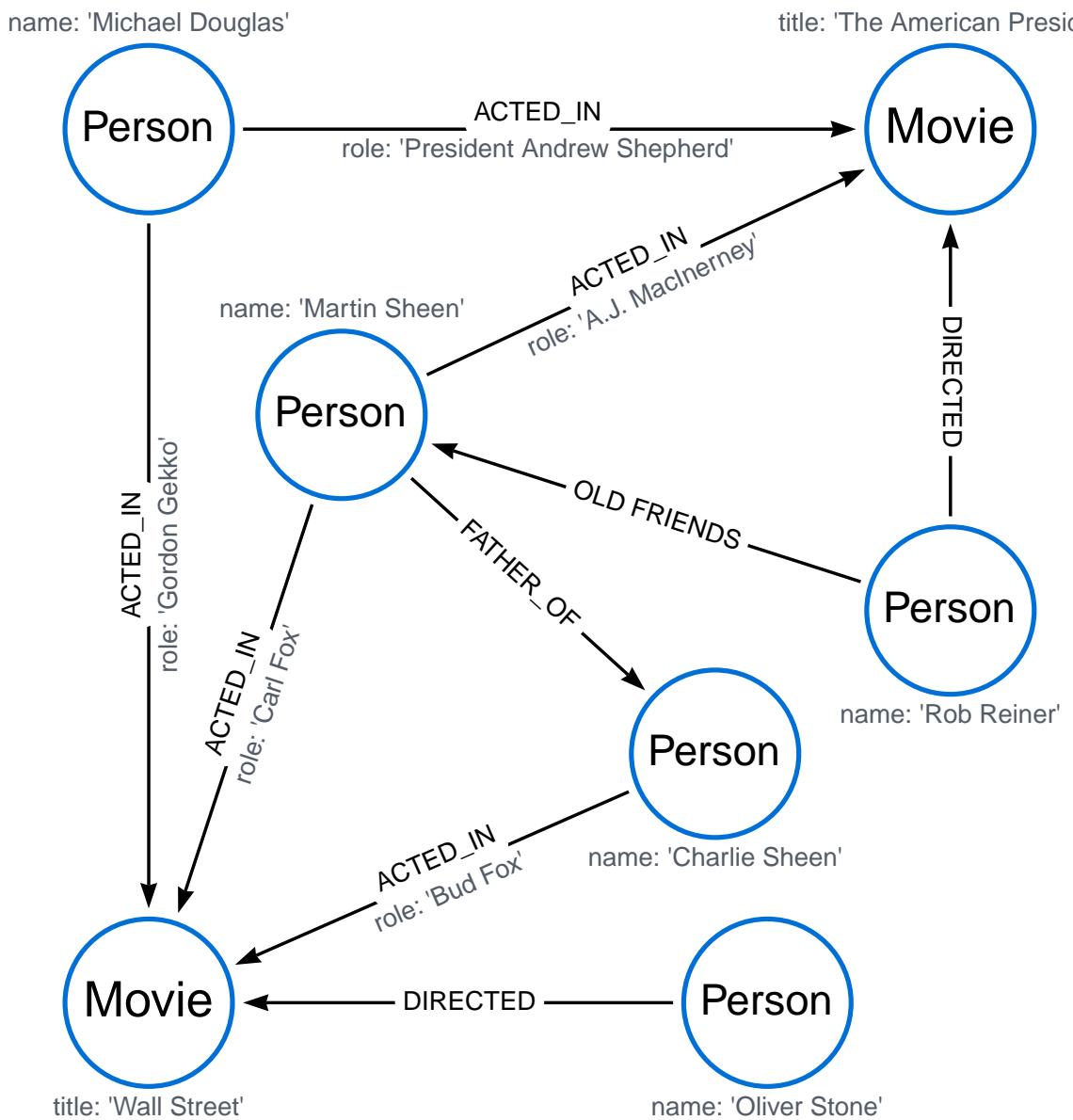
Databases occasionally contain relationship types including non-alphanumeric characters, or with spaces in them. These are created using backticks (`).

For example, the following query creates a relationship which contains a space (OLD FRIENDS) between Martin Sheen and Rob Reiner.

Query

```
MATCH
  (charlie:Person {name: 'Martin Sheen'}),
  (rob:Person {name: 'Rob Reiner'})
CREATE (rob)-[`OLD FRIENDS`]->(charlie)
```

This leads to the following graph:



Query

```

MATCH (n {name: 'Rob Reiner'})-[r: 'OLD FRIENDS']->()
RETURN type(r)
  
```

Table 136. Result

type(r)
"OLD FRIENDS"
Rows: 1

Multiple relationships

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

Query

```

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

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

Table 137. 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 as this implies a fixed length pattern.



Variable length relationships can be planned with an optimisation under certain circumstances, see [VarLength Expand Pruning](#) query plan.

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:

- **Wall Street** is found through the direct connection, whereas the other two results are found via **Michael Douglas** and **Martin Sheen** respectively.
- As this example demonstrates, variable length relationships do not impose any requirements on the intermediate nodes.

Table 138. 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, `*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 139. 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 140. Result

relationships(p)
{role:"Bud Fox"},{role:"Gordon Gekko"}
{role:"Bud Fox"},{role:"Carl Fox"}
Rows: 2

Match with properties on a variable length path

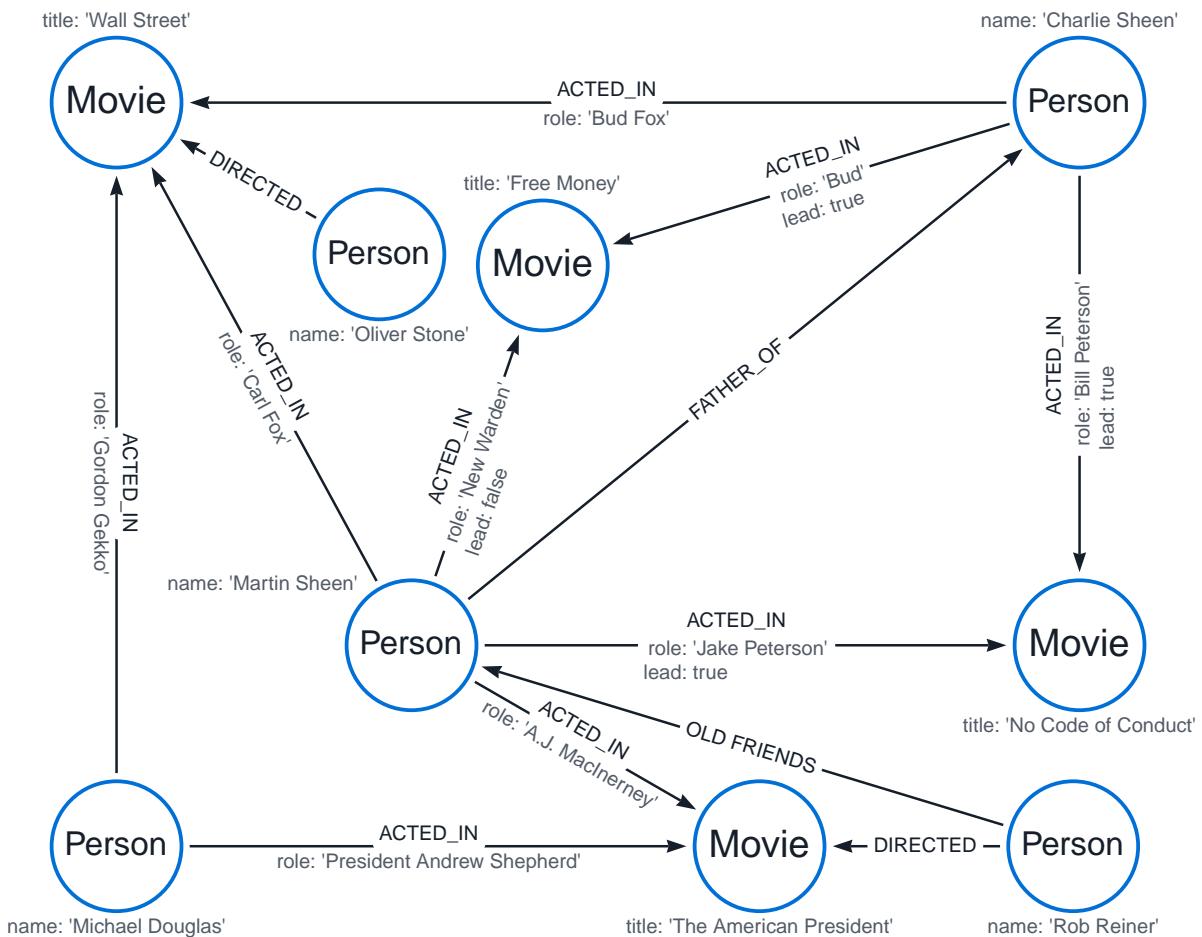
A variable length relationship with properties defined on it means that all relationships in the path must have the property set to the given value.

The following query adds two new paths between **Charlie Sheen** and his father **Martin Sheen**, where a **lead** property is added to the **ACTED_IN** relationships connecting them to the **Movie** nodes **No Code of Conduct** and **Free Money**. The query makes evident that both actors had a leading role in the movie **No Code of Conduct**, but only **Martin Sheen** had a leading role in the movie **Free Money**.

Query

```
MATCH
  (charlie:Person {name: 'Charlie Sheen'}),
  (martin:Person {name: 'Martin Sheen'})
CREATE (charlie)-[:ACTED_IN {role: 'Bud', lead: true}]->(:Movie {title: 'Free Money'})<-[:ACTED_IN {
  role: 'New Warden', lead: false}]->(martin),
  (charlie)-[:ACTED_IN {role: 'Jake Peterson', lead: true}]->(:Movie {title: 'No Code of Conduct'})<-
  [:ACTED_IN {role: 'Bill Peterson', lead: true}]->(martin)
```

This leads to the following graph:



Query

```

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

```

The above query returns the paths between **Charlie Sheen** and **Martin Sheen** where all relationships have the **lead** property set to **true**. The following graph and text are returned:

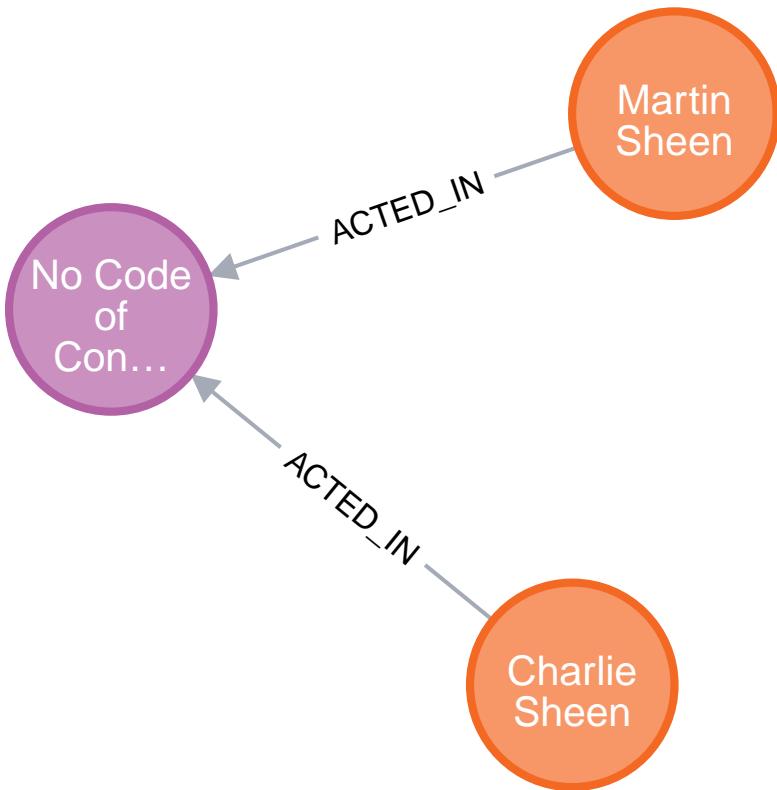


Table 141. Result

p
[{"name": "Charlie Sheen", "role": "Jake Peterson", "lead": true}, {"title": "No Code of Conduct"}, {"role": "Bill Peterson", "lead": true}, {"name": "Martin Sheen"}]
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 142. Result

x
{title: "Wall Street"}
{name: "Oliver Stone"}
{name: "Michael Douglas"}
{name: "Martin Sheen"}

```
x  
{name:"Charlie Sheen"}  
Rows: 5
```

Named paths

It is possible to introduce a named path to return or filter on a path in the pattern graph. For example:

Query

```
MATCH p = (michael {name: 'Michael Douglas'})-->()  
RETURN p
```

This query returns the following graph and text, showing the two paths starting from [Michael Douglas](#).

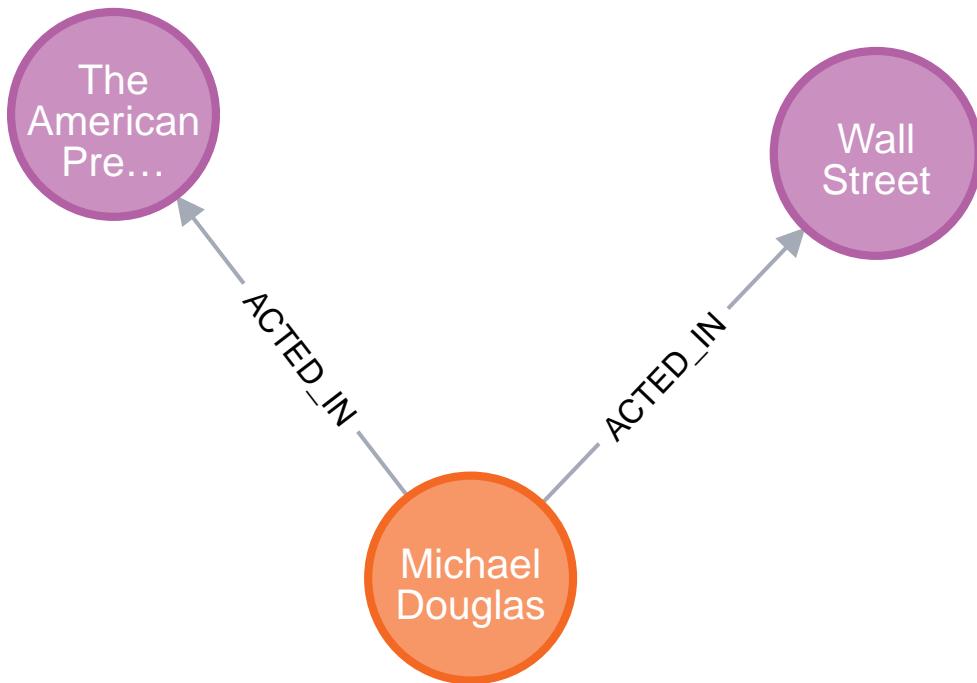


Table 143. Result

```
p  
[{"name": "Michael Douglas", "role": "Gordon Gekko", "title": "Wall Street"}, {"name": "Michael Douglas", "role": "President Andrew Shepherd", "title": "The American President"}]  
Rows: 2
```

Matching on a bound relationship

When a pattern contains a bound relationship, and that relationship pattern does not specify direction, Cypher will try to match the relationship in both directions. For example:

Query

```
MATCH (a)-[r]-(b)
WHERE split(elementId(r), ":")[2] = "0"
RETURN a, b
```

This returns the two connected nodes, once as the start node, and once as the end node

Table 144. Result

a	b
{name:"Charlie Sheen"}	{title:"Wall Street"}
{title:"Wall Street"}	{name:"Charlie Sheen"}
Rows: 2	

Shortest path

Single shortest path

Finding a single shortest path between two nodes can be done by using the `shortestPath` function.

Query

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

This query finds the shortest path between two nodes, as long as the path is max 15 relationships long. The path link (the starting node, the connecting relationships, and the end node) is defined within the parentheses. 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](#)).

The query returns the following graph and text, showing the shortest possible path between the start node (`Martin Sheen`) and the end node (`Oliver Stone`):

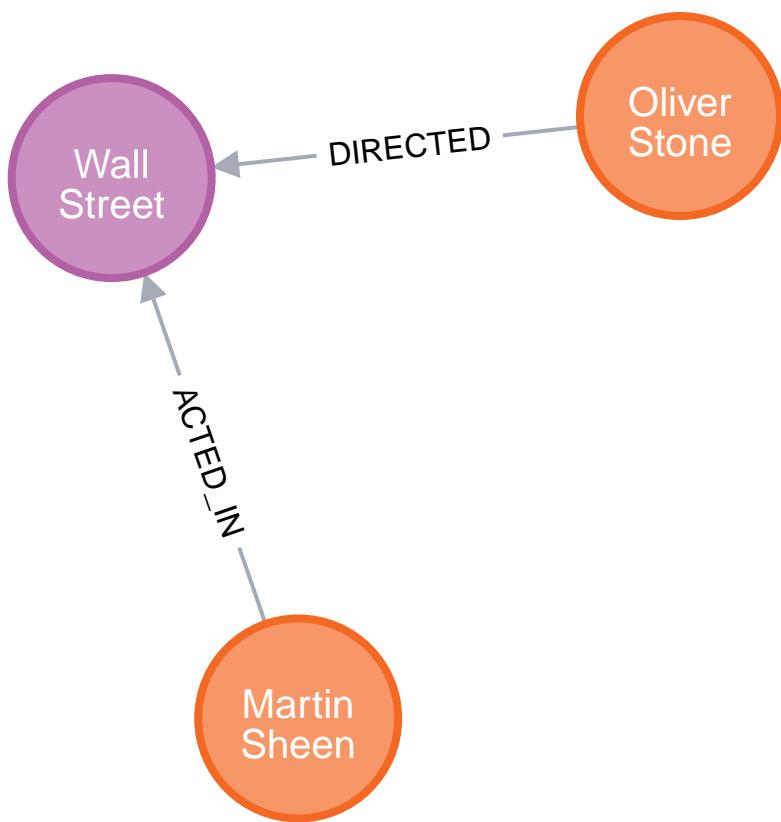


Table 145. Result

p
+[{"name": "Martin Sheen", "role": "Carl Fox"}, {"title": "Wall Street"}, {"title": "Wall Street"}, {}, {"name": "Oliver Stone"}]
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_OF')
RETURN p

```

This query will find the shortest path between `Charlie Sheen` and `Martin Sheen`, and the `WHERE` predicate will ensure that the `FATHER_OF` relationship between the two is not considered.

It returns the following graph and text:

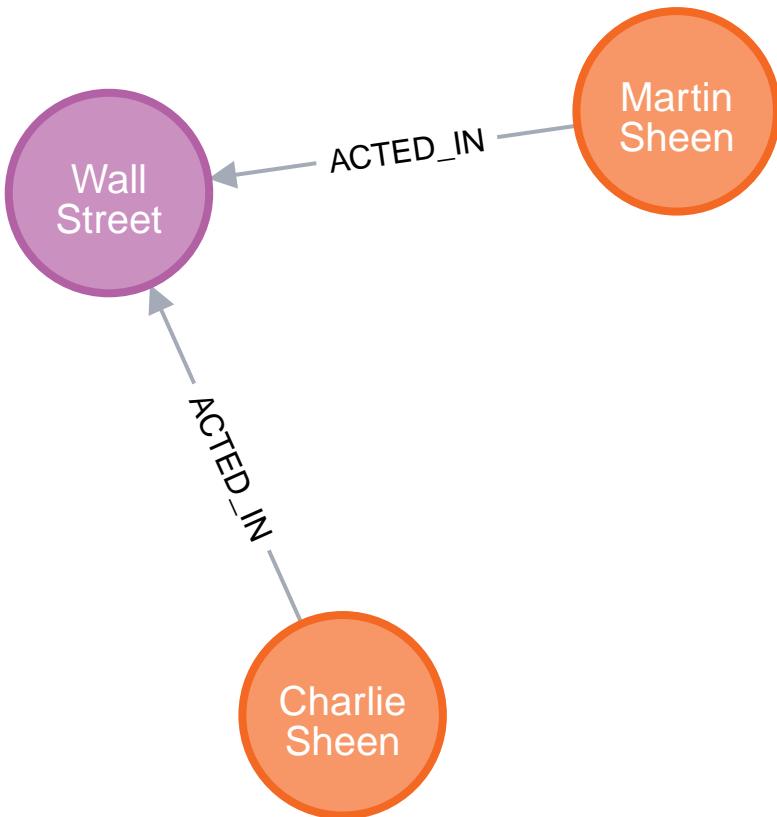


Table 146. Result

<code>p</code>
<code>[{"name": "Charlie Sheen", "role": "Bud Fox"}, {"title": "Wall Street"}, {"title": "Wall Street"}, {"role": "Carl Fox"}, {"name": "Martin Sheen"}]</code>
Rows: 1

All shortest paths

Finding all shortest paths between two nodes can be done by using the `allShortestPaths` function:

Query

```

MATCH
  (martin:Person {name: 'Martin Sheen'} ),
  (michael:Person {name: 'Michael Douglas'}),
  p = allShortestPaths((martin)-[*]-(michael))
RETURN p

```

This query finds the two shortest paths between `Martin Sheen` and `Michael Douglas`. It returns the following graph and text:

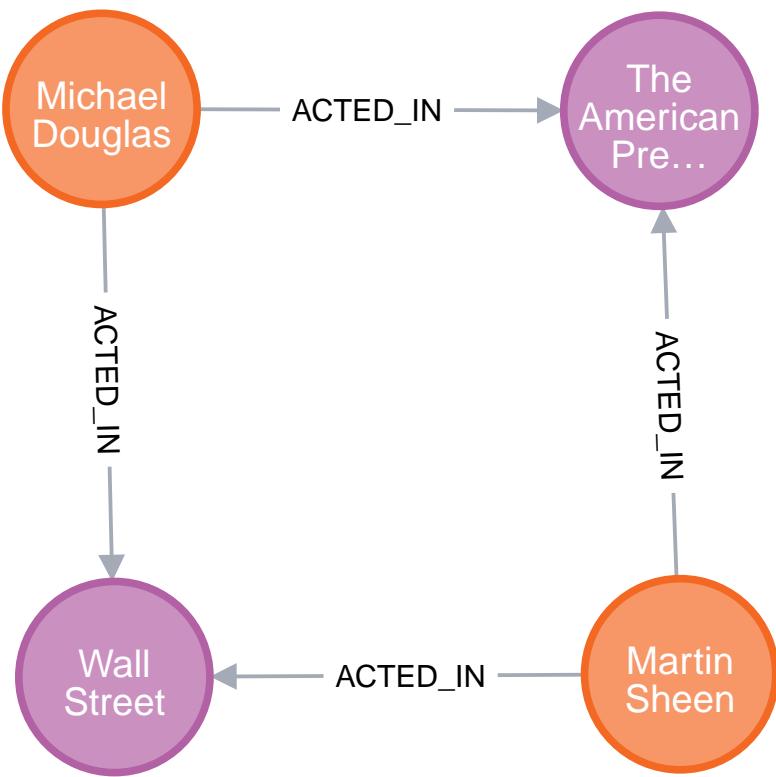


Table 147. Result

p
[{"name": "Martin Sheen", "role": "Carl Fox"}, {"title": "Wall Street", "title": "Wall Street"}, {"role": "Gordon Gekko"}, {"name": "Michael Douglas"}]
[{"name": "Martin Sheen", "role": "A.J. MacInerney"}, {"title": "The American President", "title": "The American President"}, {"role": "President Andrew Shepherd"}, {"name": "Michael Douglas"}]
Rows: 2

Get node or relationship by elementId

Node by elementId

Searching for nodes by ID can be done with the `elementId()` 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 split(elementId(n), ":")[2] = "0"
RETURN n
  
```

The corresponding node is returned.

Table 148. Result

```
n  
{name:"Charlie Sheen"}  
Rows: 1
```

Relationship by elementId

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

Query

```
MATCH ()-[r]->()  
WHERE split(elementId(r), ":")[2] = "0"  
RETURN r
```

The relationship with the elementId `0` is returned.

Table 149. Result

```
r  
{role:"Bud Fox"}  
Rows: 1
```

Multiple nodes by elementId

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

Query

```
MATCH (n)  
WHERE split(elementId(n), ":")[2] IN ["0", "3", "5"]  
RETURN n
```

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

Table 150. Result

```
n  
{name:"Charlie Sheen"}  
{name:"Oliver Stone"}  
{title:"Wall Street"}  
Rows: 3
```

OPTIONAL MATCH

The `OPTIONAL MATCH` clause is used to search for the pattern described in it, while using nulls for missing parts of the pattern.

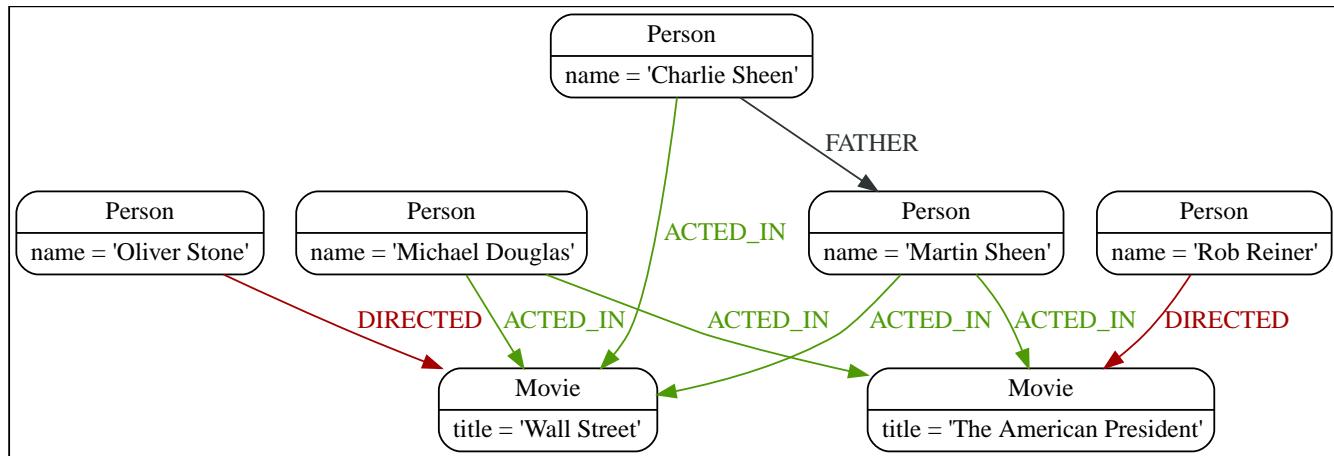
`OPTIONAL MATCH` matches patterns against your graph database, just like `MATCH` does. The difference is that if no matches are found, `OPTIONAL MATCH` will use a `null` for missing parts of the pattern. `OPTIONAL MATCH` could be considered the Cypher equivalent of the outer join in SQL.

Either the whole pattern is matched, or nothing is matched. Remember that `WHERE` is part of the pattern description, and the predicates will be considered while looking for matches, not after. This matters especially in the case of multiple (`OPTIONAL`) `MATCH` clauses, where it is crucial to put `WHERE` together with the `MATCH` it belongs to.



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

The following graph is used for the examples below:



Optional relationships

If a relationship is optional, use the `OPTIONAL MATCH` clause. This is similar to how a SQL outer join works. If the relationship is there, it is returned. If it is not, `null` is returned in its place.

Query

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

Returns `null`, since the node has no outgoing relationships.

Table 151. Result

x
<null>
Rows: 1

Properties on optional elements

Returning a property from an optional element that is `null` will also return `null`.

Query

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

Returns the element x (`null` in this query), and `null` as its name.

Table 152. Result

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

Optional typed and named relationship

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

Query

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

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

Table 153. Result

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

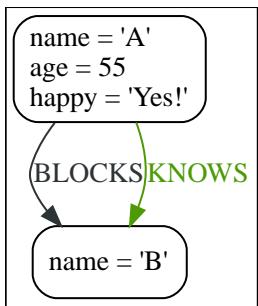
RETURN

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

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



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 154. 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'})-[:KNOWS]->(c)
RETURN r

```

The relationship is returned by the example.

Table 155. 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 156. 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 157. 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 158. 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 159. 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 160. 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 161. Result

a.age > 30	"I'm a literal"	(a)-->()
true	"I'm a literal"	[()-[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 162. 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.

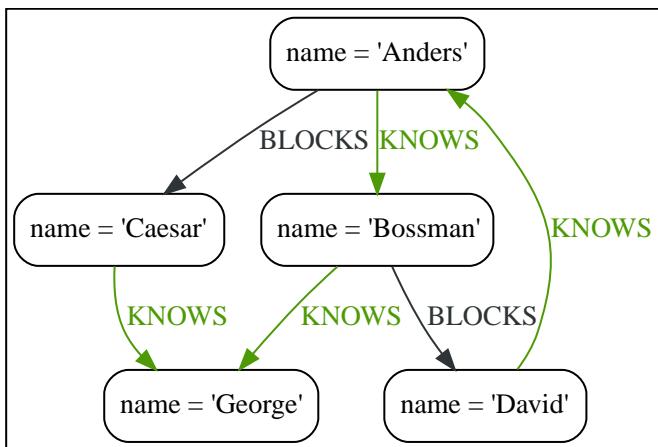
One common usage of **WITH** is to limit the number of entries passed on to other **MATCH** clauses. By combining **ORDER BY** and **LIMIT**, it is possible to get the top X entries by some criteria and then bring in additional data from the graph.

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

Table 165. 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 166. 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 167. Result

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

Limit and Filtering

It is possible to limit and filter on the same **WITH** clause. Note that the **LIMIT** clause is applied before the **WHERE** clause.

Query

```
UNWIND [1, 2, 3, 4, 5, 6] AS x
WITH x
LIMIT 5
WHERE x > 2
RETURN x
```

The limit is first applied, reducing the rows to the first 5 items in the list. The filter is then applied, reducing the final result as seen below:

Table 168. Result

x
3

x
4
5
Rows: 3

If the desired outcome is to filter and then limit, the filtering needs to occur in its own step:

Query

```
UNWIND [1, 2, 3, 4, 5, 6] AS x
WITH x
WHERE x > 2
WITH x
LIMIT 5
RETURN x
```

This time the filter is applied first, reducing the rows to consist of the list [3, 4, 5, 6]. Then the limit is applied. As the limit is larger than the total number of remaining rows, all rows are returned.

Table 169. Result

x
3
4
5
6
Rows: 4

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 170. 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 171. 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 172. 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 173. 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 empty, 'literal_that_is_not_returned'
```

Table 174. 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 175. 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 176. 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](#)
 - [Node pattern predicates](#)
 - [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
- 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
 - Relationship pattern predicates

Introduction

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

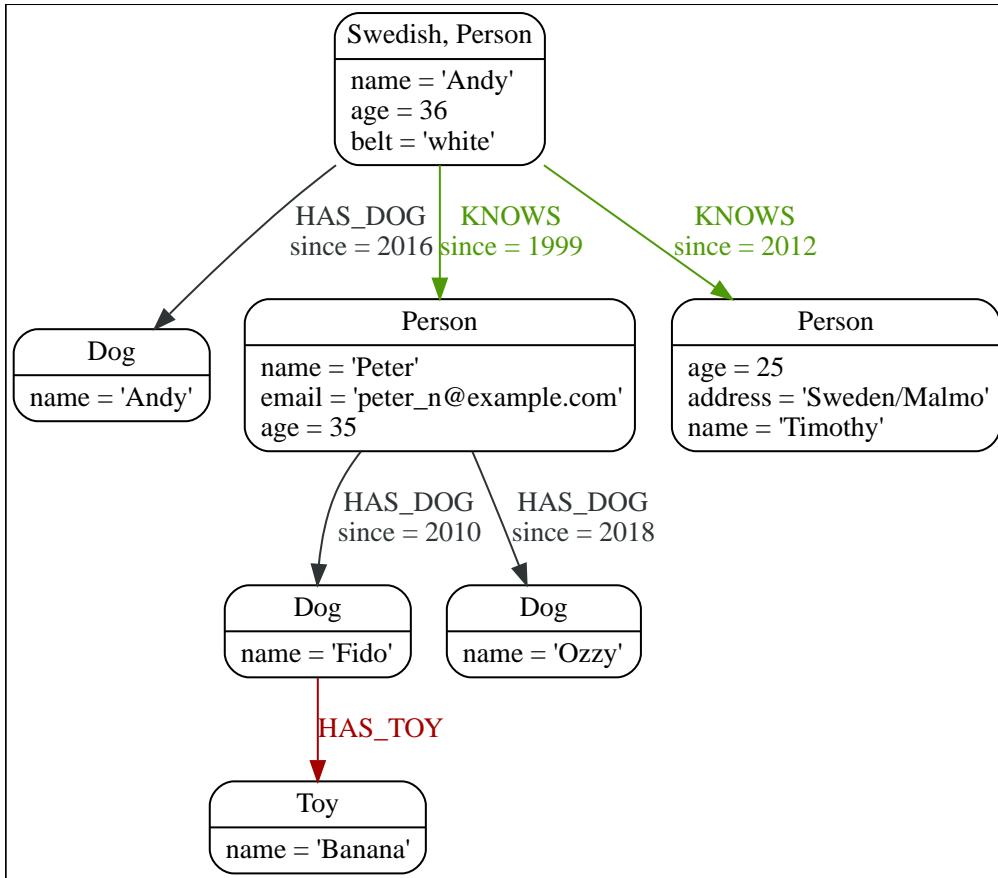
In the case of `WITH`, `WHERE` simply filters the results.

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 <code>MATCH</code> / <code>OPTIONAL MATCH</code> clauses, the predicate in <code>WHERE</code> is always a part of the patterns in the directly preceding <code>MATCH</code> / <code>OPTIONAL MATCH</code> . Both results and performance may be impacted if the <code>WHERE</code> is put inside the wrong <code>MATCH</code> clause.
	<code>Indexes</code> may be used to optimize queries using <code>WHERE</code> in a variety of cases.

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
(:Swedish:Person {name: 'Andy', age: 36, belt: 'white'}),
(:timothy:Person {name: 'Timothy', age: 25, address: 'Sweden/Malmo'}),
(:peter:Person {name: 'Peter', age: 35, email: 'peter_n@example.com'}),
(:andy)-[:KNOWS {since: 2012}]->(:timothy),
(:andy)-[:KNOWS {since: 1999}]->(:peter),
(:andy)-[:HAS_DOG {since: 2016}]->(:Dog {name:'Andy'}),
(:fido:Dog {name:'Fido'})-<[:HAS_DOG {since: 2010}]->(:peter)-[:HAS_DOG {since: 2018}]->(:Dog {name:'Ozzy'}),
(:fido)-[:HAS_TOY]->(:Toy {name:'Banana'})
    
```

Basic usage

Node pattern predicates

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

Example 58. WHERE

Query

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

Table 177. 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.

Example 59. WHERE

Query

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

Table 178. Result

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

Boolean operations

You can use the boolean operators `AND`, `OR`, `XOR` and `NOT`. See [Working with null](#) for more information on how this works with `null`.

Query

```
MATCH (n:Person)
WHERE n.name = 'Peter' XOR (n.age < 30 AND n.name = 'Timothy') OR NOT (n.name = 'Timothy' OR n.name = 'Peter')
RETURN
  n.name AS name,
  n.age AS age
ORDER BY name
```

Table 179. Result

name	age
"Andy"	36

name	age
"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 180. 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 181. 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 182. 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 183. 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.

Table 184. Result

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

Usage with `WITH`

As `WHERE` is not considered a clause in its own right, its scope is not limited by a `WITH` directly before it.

Query

```
MATCH (n:Person)
WITH n.name as name
WHERE n.age = 25
RETURN name
```

Table 185. Result

name
"Timothy"
Rows: 1

The name for the 'Timothy' node is returned because the `WHERE` clause still acts as a filter on the `MATCH`. The `WITH` reduces the scope for the rest of the query moving forward. In this case 'name' is now the only variable in scope for the `RETURN` clause.

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 186. 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 187. 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 188. Result

n.name	n.age
"Peter"	35
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 189. 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 an example of a regular expression flag given at the beginning of a pattern, see the [case-insensitive regular expression](#) section.

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 190. 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 191. 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
\t	Tab	Tab
\\\t	\t	Tab
\b	Backspace	Backspace

String literal sequence	Resulting Regex sequence	Regex match
\b	\b	Word boundary
\n	\n	NewLine
\n	\n	Newline
\r	\r	Carriage return
\r	\r	Carriage return
\f	\f	Form feed
\f	\f	Form feed
\'	\'	Single quote
\"	\"	Double quote
\\\	\\\	Backslash
\\\\	\\\\	Backslash
\uxxxx	Unicode UTF-16 code point (4 hex digits must follow the \u)	Unicode UTF-16 code point (4 hex digits must follow the \u)
\uxxxx \uxxxx		Unicode UTF-16 code point (4 hex digits must follow the \u)



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 192. 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 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 193. 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 194. 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]-(m{name: 'Timothy'})
RETURN n.name, m.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 195. Result

n.name	m.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 196. Result

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

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 197. 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 198. 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 199. 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 200. 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 201. 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 202. Result

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

Pattern element predicates

WHERE clauses can be added to [pattern elements](#) in order to specify additional constraints.

Relationship pattern predicates

WHERE can also appear inside a relationship pattern in a MATCH clause.

Example 60. WHERE

Query

```
WITH 2000 AS minYear
MATCH (a:Person)-[r:KNOWS WHERE r.since < minYear]->(b:Person)
RETURN r.since
```

Table 203. Result

r.since
1999
Rows: 1

However, it cannot be used inside of variable length relationships, as this would lead to an error.

Example 61. WHERE

For example:

Query

```
WITH 2000 AS minYear
MATCH (a:Person)-[r:KNOWS*1..3 WHERE r.since > b.yearOfBirth]->(b:Person)
RETURN r.since
```

Error message

```
Relationship pattern predicates are not supported for variable-length relationships.
```

Putting predicates inside a relationship pattern can help with readability. Please note that it is strictly equivalent to using a standalone `WHERE` sub-clause.

Example 62. WHERE

Query

```
WITH 2000 AS minYear
MATCH (a:Person)-[r:KNOWS]->(b:Person)
WHERE r.since < minYear
RETURN r.since
```

Table 204. Result

r.since
1999
Rows: 1

Relationship pattern predicates can also be used inside pattern comprehensions, where the same caveats apply.

Example 63. WHERE

Query

```
WITH 2000 AS minYear
MATCH (a:Person {name: 'Andy'})
RETURN [(a)-[r:KNOWS WHERE r.since < minYear]->(b:Person) | r.since] AS years
```

Table 205. Result

years
[1999]
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.

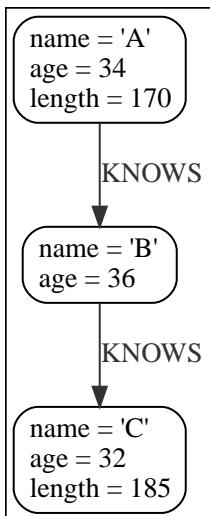
`ORDER BY` relies on comparisons to sort the output, see [Ordering and comparison of values](#). You can sort on many different values, e.g. node/relationship properties, the node/relationship ids, or on most expressions. If you do not specify what to sort on, there is a risk that the results are arbitrarily 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 206. 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 207. 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 elementId(n)
```

The nodes are returned, sorted by their internal ID.

Table 208. 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 209. 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 210. 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 211. 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 212. Result

```
names
```

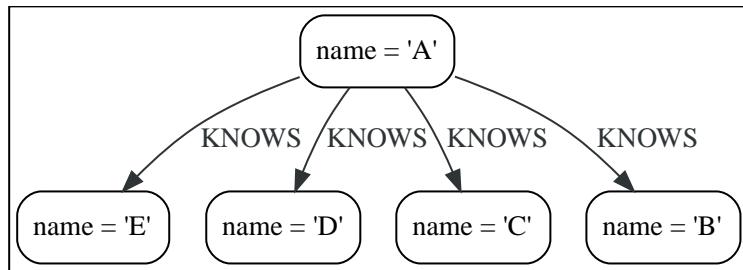
```
["C", "A", "B"]
```

```
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 213. 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 214. 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.

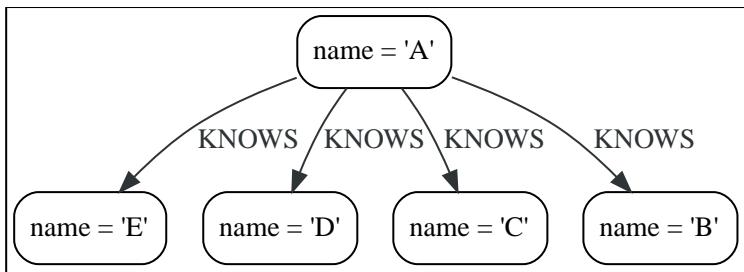
Table 215. Result

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 216. 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 217. 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 218. 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 219. 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 220. 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 221. 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 222. 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 223. 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 224. 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 225. 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 226. 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 227. 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 228. 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 229. 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 230. 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 231. Result

(empty result)
Rows: 0 Nodes created: 2 Properties set: 4

DELETE

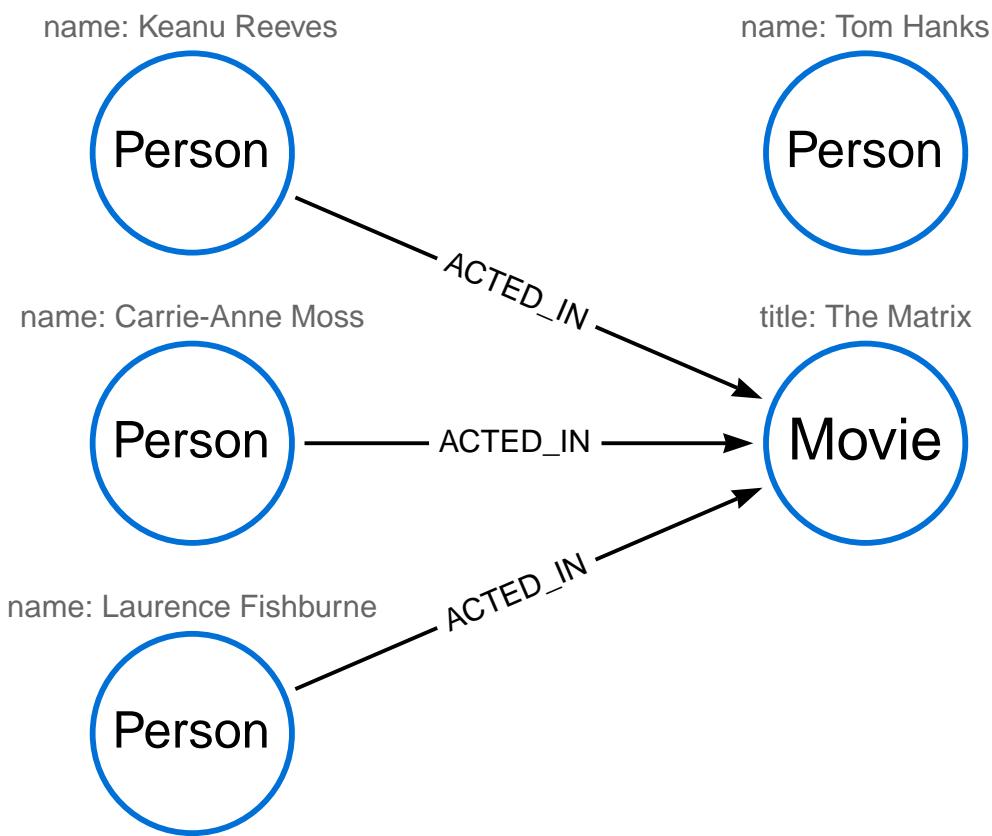
The **DELETE** clause is used to delete nodes, relationships or paths.

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 to 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

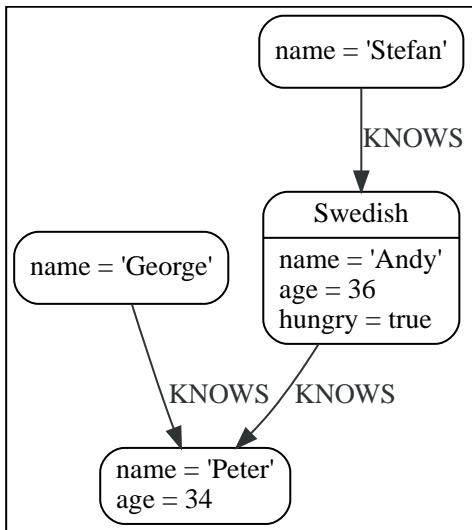
The **SET** clause is used to update labels on nodes and properties on nodes and relationships.

The **SET** clause can be used with a map — provided as a literal or a parameter — 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

Use **SET** to set a property on a node or relationship:

Query

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

The newly-changed node is returned by the query.

Table 232. Result

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

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 233. 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 234. Result

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

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 235. 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 236. 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 using the `properties()` function. This will remove all other properties on the node or relationship being copied to.

Query

```
MATCH
  (at {name: 'Andy'}),
  (pn {name: 'Peter'})
SET at = properties(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 237. 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 238. 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 239. 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 240. 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 241. 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 242. 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 243. 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 244. 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 245. 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 246. Result

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

Rows: 1
Labels added: 2

REMOVE

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

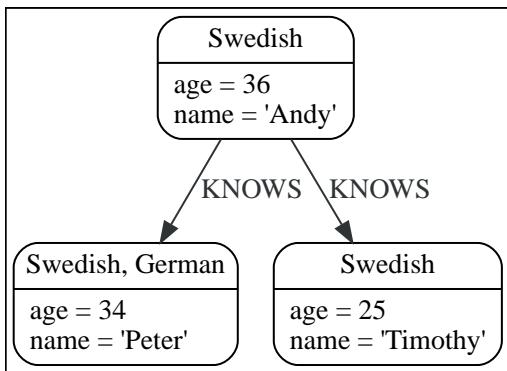


For deleting nodes and relationships, see [DELETE](#).



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 247. 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 248. 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 249. 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.

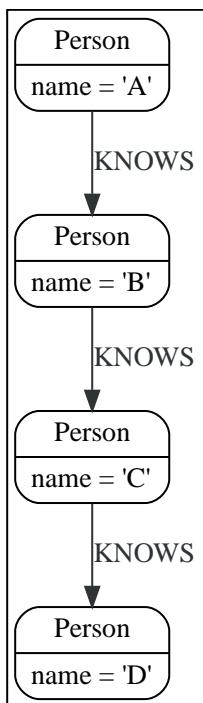
Lists and paths are key concepts in Cypher. The **FOREACH** clause can be used to update data, such as executing update commands on elements in a path, or on a list created by aggregation.

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 250. 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 property uniqueness constraints with MERGE
 - Merge using property uniqueness constraints creates a new node if no node is found
 - Merge using property uniqueness constraints matches an existing node
 - Merge with property uniqueness constraints and partial matches
 - Merge with property uniqueness 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.



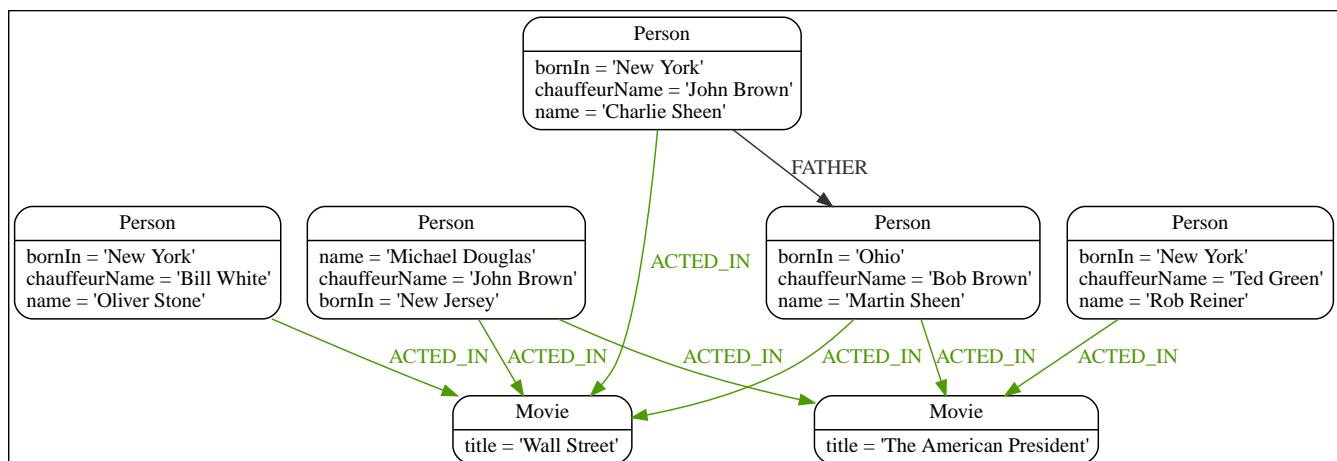
Under concurrent updates, `MERGE` only guarantees existence of the `MERGE` pattern, but not uniqueness. To guarantee uniqueness of nodes with certain properties, a [property uniqueness constraint](#) should be used. See [Using property uniqueness constraints with MERGE](#) to see how `MERGE` can be used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

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', bornIn: 'New York', chauffeurName: 'John Brown'}),
(martin:Person {name: 'Martin Sheen', bornIn: 'Ohio', chauffeurName: 'Bob Brown'}),
(michael:Person {name: 'Michael Douglas', bornIn: 'New Jersey', chauffeurName: 'John Brown'}),
(oliver:Person {name: 'Oliver Stone', bornIn: 'New York', chauffeurName: 'Bill White'}),
(rob:Person {name: 'Rob Reiner', bornIn: 'New York', chauffeurName: 'Ted Green'}),
(wallStreet:Movie {title: 'Wall Street'}),
(theAmericanPresident:Movie {title: 'The American President'}),
(charlie)-[:ACTED_IN]->(wallStreet),
(martin)-[:ACTED_IN]->(wallStreet),
(michael)-[:ACTED_IN]->(wallStreet),
(martin)-[:ACTED_IN]->(theAmericanPresident),
(michael)-[:ACTED_IN]->(theAmericanPresident),
(oliver)-[:ACTED_IN]->(wallStreet),
(rob)-[:ACTED_IN]->(theAmericanPresident),
(charlie)-[:FATHER]->(martin)
```

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 251. Result

robert	labels(robert)
{}	["Critic"]

Merge single node with properties

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

Query

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

A new node with the name '`Charlie Sheen`' will be created since not all properties matched the existing '`Charlie Sheen`' node.

Table 252. Result

charlie
{"name": "Charlie Sheen", "age": 10}

Merge single node specifying both label and property

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

Query

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

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

Table 253. Result

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

Merge single node derived from an existing node property

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

Query

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

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

Table 254. Result

person.name	person.bornIn	city
"Charlie Sheen"	"New York"	{name: "New York"}
"Martin Sheen"	"Ohio"	{name: "Ohio"}
"Michael Douglas"	"New Jersey"	{name: "New Jersey"}
"Oliver Stone"	"New York"	{name: "New York"}
"Rob Reiner"	"New York"	{name: "New York"}

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', bornIn: 'Beirut', chauffeurName: 'Eric Brown'})
ON CREATE
  SET keanu.created = timestamp()
RETURN keanu.name, keanu.created
```

The query creates the `Person` node named `Keanu Reeves`, with a `bornIn` property set to `Beirut` and a `chauffeurName` property set to `Eric Brown`. It also sets a timestamp for the `created` property.

Table 255. Result

keanu.name	keanu.created
"Keanu Reeves"	1655200898563

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 256. Result

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

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
```

Because the **Person** node named **Keanu Reeves** already exists, this query does not create a new node. Instead, it adds a timestamp on the **lastSeen** property.

Table 257. Result

keanu.name	keanu.created	keanu.lastSeen
"Keanu Reeves"	1655200902354	1674655352124

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 258. 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
"Keanu Reeves"	true	1655200903558

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 259. Result

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

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 our example graph, '[Oliver Stone](#)' and '[Rob Reiner](#)' have never worked together. When we try to [MERGE](#) a "movie between them, Neo4j will not use any of the existing movies already connected to either person. Instead, a new '[movie](#)' node is created.

Table 260. Result

movie
{}

Merge on an undirected relationship

[MERGE](#) can also be used with an undirected relationship. When it needs to create a new one, it will pick a direction.

Query

```

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

As '[Charlie Sheen](#)' and '[Oliver Stone](#)' do not know each other this [MERGE](#) query will create a [KNOWS](#) relationship between them. The direction of the created relationship is arbitrary.

Table 261. Result

r
{}

Merge on a relationship between two existing nodes

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

Query

```

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

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

Table 262. Result

person.name	person.bornIn	city
"Charlie Sheen"	"New York"	{name: "New York"}
"Martin Sheen"	"Ohio"	{name: "Ohio"}

person.name	person.bornIn	city
"Michael Douglas"	"New Jersey"	{name: "New Jersey"}
"Oliver Stone"	"New York"	{name: "New York"}
"Rob Reiner"	"New York"	{name: "New York"}
"Keanu Reeves"	"Beirut"	{name: "Beirut"}

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 263. Result

person.name	person.chauffeurName	chauffeur
"Charlie Sheen"	"John Brown"	{name: "John Brown"}
"Martin Sheen"	"Bob Brown"	{name: "Bob Brown"}
"Michael Douglas"	"John Brown"	{name: "John Brown"}
"Oliver Stone"	"Bill White"	{name: "Bill White"}
"Rob Reiner"	"Ted Green"	{name: "Ted Green"}
"Keanu Reeves"	"Eric Brown"	{name: "Eric Brown"}

Using property uniqueness constraints with `MERGE`

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

For example, given two property uniqueness constraints on `:Person(id)` and `:Person(ssn)`, a query such as `MERGE (n:Person {id: 12, ssn: 437})` will fail, if there are two different nodes (one with `id` 12 and one

with `ssn` 437), or if there is only one node with only one of the properties. In other words, there must be exactly one node that matches the pattern, or no matching nodes.

Note that the following examples assume the existence of property uniqueness 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 property uniqueness constraints creates a new node if no node is found

Merge using property uniqueness 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 264. Result

laurence.name
"Laurence Fishburne"

Merge using property uniqueness constraints matches an existing node

Merge using property uniqueness 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 265. Result

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

Merge with property uniqueness constraints and partial matches

Merge using property uniqueness constraints fails when finding partial matches.

Query

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

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

Error message

```
Node already exists with label `Person` and property `name` = 'Michael Douglas'
```

If we want to give Michael Douglas the role of Gordon Gekko, we can use the **SET** clause instead:

Query

```
MERGE (michael:Person {name: 'Michael Douglas'})
SET michael.role = 'Gordon Gekko'
```

Result

```
Set 1 property
```

Merge with property uniqueness constraints and conflicting matches

Merge using property uniqueness constraints fails when finding conflicting matches.

Query

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

While there is a matching unique 'oliver' node with the name 'Oliver Stone', there is also another unique node with the role of 'Gordon Gekko' and **MERGE** fails to match.

Error message

```
Node already exists with label `Person` and property `name` = 'Oliver Stone'
```

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 266. Result

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

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*.

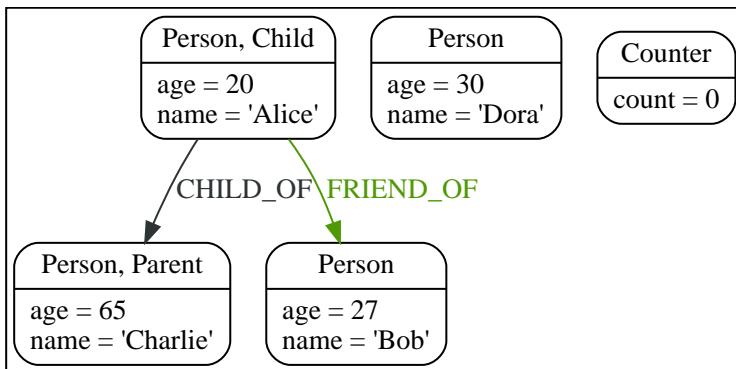
A subquery is evaluated for each incoming input row. Every output row of a returning subquery is combined with the input row to build the result of the subquery. That means that a returning subquery will influence the number of rows. If the subquery does not return any rows, there will be no rows available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



Semantics

A `CALL` clause is executed once for each incoming row.

Example 64. Execute for each incoming row

The `CALL` clause executes three times, one for each row that the `UNWIND` clause outputs.

Query

```
UNWIND [0, 1, 2] AS x
CALL {
  RETURN 'hello' AS innerReturn
}
RETURN innerReturn
```

Result

Rows: 3

innerReturn
'hello'
'hello'
'hello'

Each execution of a `CALL` clause can observe changes from previous executions.

Example 65. Observe changes from previous execution

Query

```
UNWIND [0, 1, 2] AS x
CALL {
  MATCH (n:Counter)
    SET n.count = n.count + 1
  RETURN n.count AS innerCount
}
WITH innerCount
MATCH (n:Counter)
RETURN
  innerCount,
  n.count AS totalCount
```

Result

Set Properties: 3
Rows: 3

innerCount	totalCount
1	3
2	3
3	3

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 267. 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).



The order in which subqueries are executed is not defined. If a query result depends on the order of execution of subqueries, an `ORDER BY` clause should precede the `CALL` clause.

Example 66. The order in which subqueries are executed

This query creates a linked list of all :Person nodes in order of ascending age.

The `CALL` clause is relying on the incoming row ordering to ensure that a correctly linked list is created, thus the incoming rows must be ordered with a preceding `ORDER BY` clause.

Query

```
MATCH (person:Person)
WITH person ORDER BY person.age ASC LIMIT 1
    SET person>ListHead
WITH *
MATCH (next: Person)
    WHERE NOT next>ListHead
WITH next ORDER BY next.age
CALL {
    WITH next
    MATCH (current>ListHead)
    REMOVE current>ListHead
    SET next>ListHead
    CREATE(current)-[r:IS_YOUNGER_THAN]->(next)
    RETURN current AS from, next AS to
}
RETURN
    from.name AS name,
    from.age AS age,
    to.name AS closestOlderName,
    to.age AS closestOlderAge
```

Result

```
Added Labels: 4
Created Relationships: 3
Removed Labels: 3
Rows: 3
```

name	age	closestOlderName	closestOlderAge
'Alice'	20	'Bob'	27
'Bob'	27	'Dora'	30
'Dora'	30	'Charlie'	65

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 268. Result

p.name	p.age
"Alice"	20
"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 269. 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 270. Result

p.name	friend
"Alice"	"Bob"
"Bob"	"Alice"
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 271. 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 272. Result

count(*)
4

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 273. Result

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

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 274. 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](#).

Deleting a large volume of nodes

Using `CALL { ... } IN TRANSACTIONS` is the recommended way of deleting a large volume of nodes.

Example 67. DETACH DELETE

Query

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

Table 275. Result

(empty result)

Rows: 0

Nodes deleted: 5

Relationships deleted: 2

Transactions committed: 1



The `CALL { ... } IN TRANSACTIONS` subquery is handled by the database so as to ensure optimal performance. Modifying the subquery may result in `OutOfMemory` exceptions for sufficiently large datasets.

Example 68. DETACH DELTE

The `CALL { ... } IN TRANSACTIONS` subquery should not be modified.

Any necessary filtering can be done before the subquery.

Query

```
MATCH (n:Label) WHERE n.prop > 100
CALL {
  WITH n
  DETACH DELETE n
} IN TRANSACTIONS
```

Table 276. Result

(empty result)

Rows: 0

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 277. 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 278. 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 279. 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 1
    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 1
    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 280. 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 281. Result

label
"User"
"Administrator"
Rows: 2

View the signature for a procedure

The `SHOW PROCEDURES` command can return the `name`, `signature`, and `description` for all procedures.

The default outputs for `SHOW PROCEDURES` are `name`, `description`, `mode`, and `worksOnSystem`. To get the signature, make sure to use the `YIELD` clause.

Example 69. SHOW PROCEDURES

The following query return the signature for a particular procedure:

Query

```
SHOW PROCEDURES YIELD name, signature
WHERE name = 'dbms.listConfig'
RETURN signature
```

Table 282. 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 all queries in the union.

The number and the names of the columns must be identical in all queries combined by using `UNION`.

To keep all the result rows, use `UNION ALL`. Using just `UNION` will combine and remove duplicates from the result set.

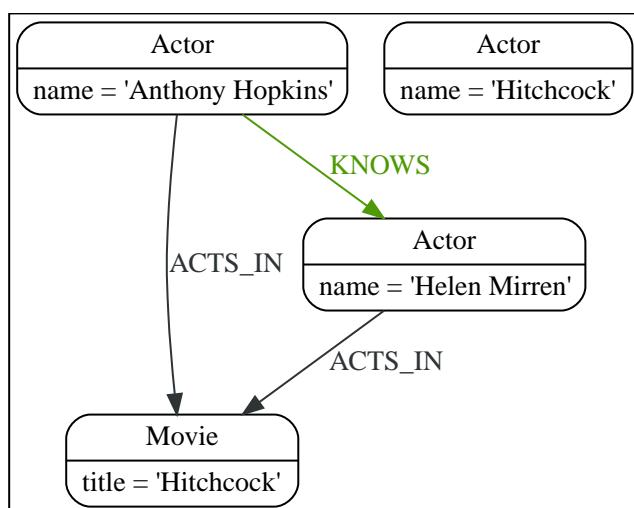
If any of the queries in a UNION contain updates, the order of queries in the UNION is relevant.



Any clause before the UNION cannot observe writes made by a clause after the UNION.

Any clause after UNION can observe all writes made by a clause before the UNION.

For details see [clause composition in queries with UNION](#) for details.



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 283. 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 284. Result

name
"Anthony Hopkins"
"Helen Mirren"
"Hitchcock"
Rows: 3

USE

The `USE` clause determines which graph a query, or query part, is executed against. It is supported for queries and schema commands.

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.

Composite database syntax

When running queries against a **composite 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 composite database, the **USE** clause must only refer to graphs that are part of the current composite database.

Examples

Query a graph

In this example it is assumed that the DBMS contains a database named **myDatabase**:

Query

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

Query a composite database constituent graph

In this example it is assumed that the DBMS contains a composite database named **myComposite**, which includes an alias named **myConstituent**:

Query

```
USE myComposite.myConstituent
MATCH (n) RETURN n
```

Query a composite database constituent graph dynamically

The built-in function `graph.byName()` can be used in the `USE` clause to resolve a constituent graph from a string value containing the qualified name of a constituent.

This example uses a composite database named `myComposite` that includes an alias named `myConstituent`:

Query

```
USE graph.byName('myComposite.myConstituent')
MATCH (n) RETURN n
```

The argument can be any expression that evaluates to the name of a constituent graph - for example a parameter:

Query

```
USE graph.byName($graphName)
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 CALL { ... } IN TRANSACTIONS](#)
- [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 clause `CALL { ... } IN TRANSACTIONS`, see [:clauses/call-subquery.pdf](#).

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`.

`server.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 `server.directories.import` directory. For example, a file URL will typically look like `file:///myfile.csv` or `file:///myproject/myfile.csv`.

- When using `file:/// URLs`, spaces and other non-alphanumeric characters need to be URL encoded.
^[9]
- If `server.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 `server.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_quoteEscaping` 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

```
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), `CALL { ... } IN TRANSACTIONS` can be used to instruct Neo4j to commit a transaction after a number of rows. This reduces the memory overhead of the transaction state.



The query clause `CALL { ... } IN TRANSACTIONS` is only allowed in `implicit (auto-commit or :auto) transactions`. For more information, see [:clauses/call-subquery.pdf](#).

artists.csv

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

Query

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

Result

```
+-----+  
| No data returned. |  
+-----+  
Nodes created: 4  
Properties set: 8  
Labels added: 4  
Transactions committed: 1
```

Setting the rate of CALL IN TRANSACTIONS

You can set the number of rows as in the example, where it is set to **500** rows.

artists.csv

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

Query

```
LOAD CSV FROM 'file:///artists.csv' AS line  
CALL {  
    WITH line  
    CREATE (:Artist {name: line[1], year: toInteger(line[2])})  
} IN TRANSACTIONS OF 500 ROWS
```

Result

```
+-----+  
| No data returned. |  
+-----+  
Nodes created: 4  
Properties set: 8  
Labels added: 4  
Transactions committed: 1
```

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 temporary 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 285. List functions output

Column	Description
<code>name</code>	The name of the function. Default output
<code>category</code>	The function category, for example <code>scalar</code> or <code>string</code> . Default output
<code>description</code>	The function description. Default output
<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



The syntax descriptions use [the style](#) from access control.

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 286. 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 287. 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 288. 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 289. 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 290. List procedures output

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

Column	Description
<code>worksOnSystem</code>	Whether the procedure can be run on the <code>system</code> database or not. Default output
<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



The syntax descriptions use the style from access control.

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 291. 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.createLabel"	"Create a label"	"WRITE"	false
"db.createProperty"	"Create a Property"	"WRITE"	false
"db.createRelationshipType"	"Create a RelationshipType"	"WRITE"	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.listAvailableAnalyzers"	"List the available analyzers that the full-text indexes can be configured with."	"READ"	true

name	description	mode	worksOnSystem
"db.index.fulltext.queryNodes"	"Query the given full-text index. Returns the matching nodes, and their Lucene query score, ordered by score. Valid keys for the options map are: 'skip' to skip the top N results; 'limit' to limit the number of results returned; 'analyzer' to use the specified analyzer as search analyzer for this query."	"READ"	true
"db.index.fulltext.queryRelationships"	"Query the given full-text index. Returns the matching relationships, and their Lucene query score, ordered by score. Valid keys for the options map are: 'skip' to skip the top N results; 'limit' to limit the number of results returned; 'analyzer' to use the specified analyzer as search analyzer for this query."	"READ"	true
"db.info"	"Provides information regarding the database."	"READ"	true
"db.labels"	"List all available labels in the database."	"READ"	true
"db.listLocks"	"List all locks in the database."	"DBMS"	true
"db.ping"	"This procedure can be used by client side tooling to test whether they are correctly connected to a database. The procedure is available in all databases and always returns true. A faulty connection can be detected by not being able to call this procedure."	"READ"	true

Rows: 15

The above table only displays the first 15 results of the query. For a full list of all built-in procedures in Neo4j, visit the [Operations Manual → List of procedures](#).

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 292. Result

name	admin
"db.clearQueryCaches"	true
"db.listLocks"	true
"db.prepareForReplanning"	true

name	admin
"db.stats.clear"	true
"db.stats.collect"	true
"db.stats.retrieve"	true
"db.stats.retrieveAllAnonymized"	true
"db.stats.status"	true
"db.stats.stop"	true
+"dbms.checkConfigValue"	true
"dbms.cluster.checkConnectivity"	true
"dbms.cluster.cordonServer"	true
"dbms.cluster.readReplicaToggle"	true
"dbms.cluster.uncordonServer"	true
"dbms.listConfig"	true

Rows: 15

The above table only displays the first 15 results of the query. For a full list of all procedures which require `admin` privileges in Neo4j, visit the [Operations Manual → List of procedures](#).

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 for using the `EXECUTABLE` clause. The first option is to filter for the current user:

Query

```
SHOW PROCEDURES EXECUTABLE BY CURRENT USER YIELD *
```

Table 293. 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>

name	description	rolesExecution	rolesBoostedExecution
"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.clearQueryCaches"	"Clears all query caches."	<null>	<null>
"db.createLabel"	"Create a label"	<null>	<null>
"db.createProperty"	"Create a Property"	<null>	<null>
"db.createRelationshipType"	"Create a RelationshipType"	<null>	<null>
"db.index.fulltext.awaitEventuallyConsistentIndexRefresh"	"Wait for the updates from recently committed transactions to be applied to any eventually-consistent full-text indexes."	<null>	<null>
"db.index.fulltext.listAvailableAnalyzers"	"List the available analyzers that the full-text indexes can be configured with."	<null>	<null>
"db.index.fulltext.queryNodes"	"Query the given full-text index. Returns the matching nodes, and their Lucene query score, ordered by score. Valid keys for the options map are: 'skip' to skip the top N results; 'limit' to limit the number of results returned; 'analyzer' to use the specified analyzer as search analyzer for this query."	<null>	<null>
"db.index.fulltext.queryRelationships"	"Query the given full-text index. Returns the matching relationships, and their Lucene query score, ordered by score. Valid keys for the options map are: 'skip' to skip the top N results; 'limit' to limit the number of results returned; 'analyzer' to use the specified analyzer as search analyzer for this query."	<null>	<null>
"db.info"	"Provides information regarding the database."	<null>	<null>
"db.labels"	"List all available labels in the database."	<null>	<null>
"db.listLocks"	"List all locks in the database."	<null>	<null>
"db.ping"	"This procedure can be used by client side tooling to test whether they are correctly connected to a database. The procedure is available in all databases and always returns true. A faulty connection can be detected by not being able to call this procedure."	<null>	<null>
Rows: 15			

The above table only displays the first 15 results of the query. Note that the two `roles` columns are empty

due to missing the `SHOW ROLE` privilege. Also note that the following columns are not present in the table: `mode`, `worksOnSystem`, `signature`, `argumentDescription`, `returnDescription`, `admin`, and `options`.

The second option for using the `EXECUTABLE` clause is to filter the list to only contain procedures executable by a specific user. The below example shows the procedures available to the user `jake`, who has been granted the `EXECUTE PROCEDURE dbms.*` privilege by the `admin` of the database. (More information about DBMS `EXECUTE` privilege administration can be found [here](#)).

Query

```
SHOW PROCEDURES EXECUTABLE BY jake
```

Table 294. Result

name	description	mode	worksOnSystem
"dbms.cluster.protocols"	"Overview of installed protocols."	"DBMS"	true
"dbms.cluster.routing.getRoutingTable"	"Returns the advertised bolt capable endpoints for a given database, divided by each endpoint's capabilities. For example an endpoint may serve read queries, write queries and/or future getRoutingTable requests."	"DBMS"	true
"dbms.components"	"List DBMS components and their versions."	"DBMS"	true
"dbms.info"	"Provides information regarding the DBMS."	"DBMS"	true
"dbms.killConnection"	"Kill network connection with the given connection id."	"DBMS"	false
"dbms.killConnections"	"Kill all network connections with the given connection ids."	"DBMS"	true
"dbms.listActiveLocks"	"List the active lock requests granted for the transaction executing the query with the given query id."	"DBMS"	true
"dbms.listCapabilities"	"List capabilities"	"DBMS"	true
"dbms.listConnections"	"List all accepted network connections at this instance that are visible to the user."	"DBMS"	true
"dbms.listPools"	"List all memory pools, including sub pools, currently registered at this instance that are visible to the user."	"DBMS"	true
"dbms.queryJmx"	"Query JMX management data by domain and name. For instance, \"*:*\""	"DBMS"	true
"dbms.routing.getRoutingTable"	"Returns the advertised bolt capable endpoints for a given database, divided by each endpoint's capabilities. For example an endpoint may serve read queries, write queries and/or future getRoutingTable requests."	"DBMS"	true
"dbms.showCurrentUser"	"Shows the current user."	"DBMS"	true

name	description	mode	worksOnSystem
Rows: 13			

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

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](#)]
- Graph 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 70. 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
any()	any(variable :: VARIABLE IN list :: LIST OF ANY? WHERE predicate :: ANY?) :: (BOOLEAN?)	Returns true if the predicate holds for at least one element in the given list.
exists()	exists(input :: ANY?) :: (BOOLEAN?)	Returns <code>true</code> if a match for the pattern exists in the graph.
isEmpty()	isEmpty(input :: LIST? OF ANY?) :: (BOOLEAN?)	Checks whether a list is empty.
	isEmpty(input :: MAP?) :: (BOOLEAN?)	Checks whether a map is empty.
	isEmpty(input :: STRING?) :: (BOOLEAN?)	Checks whether a string is empty.
none()	none(variable :: VARIABLE IN list :: LIST OF ANY? WHERE predicate :: ANY?) :: (BOOLEAN?)	Returns true if the predicate holds for no element in the given list.
single()	single(variable :: VARIABLE IN list :: LIST OF ANY? WHERE predicate :: ANY?) :: (BOOLEAN?)	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
coalesce()	coalesce(input :: ANY?) :: (ANY?)	Returns the first non-null value in a list of expressions.
endNode()	endNode(input :: RELATIONSHIP?) :: (NODE?)	Returns the end node of a relationship.
head()	head(list :: LIST? OF ANY?) :: (ANY?)	Returns the first element in a list.
id()	id(input :: NODE?) :: (INTEGER?)	Deprecated Returns the id of a node. Replaced by <code>elementId()</code>
	id(input :: RELATIONSHIP?) :: (INTEGER?)	Deprecated Returns the id of a relationship. Replaced by <code>elementId()</code> .
last()	last(list :: LIST? OF ANY?) :: (ANY?)	Returns the last element in a list.
length()	length(input :: PATH?) :: (INTEGER?)	Returns the length of a path.
properties()	properties(input :: MAP?) :: (MAP?)	Returns a map containing all the properties of a map.
	properties(input :: NODE?) :: (MAP?)	Returns a map containing all the properties of a node.
	properties(input :: RELATIONSHIP?) :: (MAP?)	Returns a map containing all the properties of a relationship.
randomUUID()	randomUUID() :: (STRING?)	Generates a random UUID.

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>isNaN()</code>	<code>isNaN(input :: FLOAT?) :: (BOOLEAN?)</code>	Returns <code>true</code> if the floating point number is <code>NaN</code> .
	<code>isNaN(input :: INTEGER?) :: (BOOLEAN?)</code>	Returns <code>true</code> if the integer number is <code>NaN</code> .
<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 <code>HALF_UP</code> .
	<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
<code>sign()</code>	<code>sign(input :: FLOAT?) :: (INTEGER?)</code>	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.
	<code>sign(input :: INTEGER?) :: (INTEGER?)</code>	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
<code>e()</code>	<code>e() :: (FLOAT?)</code>	Returns the base of the natural logarithm, e.
<code>exp()</code>	<code>exp(input :: FLOAT?) :: (FLOAT?)</code>	Returns e^n , where e is the base of the natural logarithm, and n is the value of the argument expression.
<code>log()</code>	<code>log(input :: FLOAT?) :: (FLOAT?)</code>	Returns the natural logarithm of a number.
<code>log10()</code>	<code>log10(input :: FLOAT?) :: (FLOAT?)</code>	Returns the common logarithm (base 10) of a number.
<code>sqrt()</code>	<code>sqrt(input :: FLOAT?) :: (FLOAT?)</code>	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
<code>acos()</code>	<code>acos(input :: FLOAT?) :: (FLOAT?)</code>	Returns the arccosine of a number in radians.
<code>asin()</code>	<code>asin(input :: FLOAT?) :: (FLOAT?)</code>	Returns the arcsine of a number in radians.
<code>atan()</code>	<code>atan(input :: FLOAT?) :: (FLOAT?)</code>	Returns the arctangent of a number in radians.
<code>atan2()</code>	<code>atan2(y :: FLOAT?, x :: FLOAT?) :: (FLOAT?)</code>	Returns the arctangent2 of a set of coordinates in radians.
<code>cos()</code>	<code>cos(input :: FLOAT?) :: (FLOAT?)</code>	Returns the cosine of a number.
<code>cot()</code>	<code>cot(input :: FLOAT?) :: (FLOAT?)</code>	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(milliseconds :: 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 LOAD CSV is using.
<code>linenumber()</code>	<code>linenumber() :: (INTEGER?)</code>	Returns the line number that LOAD CSV is currently using.

Graph functions

Graph functions provide information about the constituent graphs in composite databases.

Function	Signature	Description
<code>graph.names()</code>	<code>graph.names() :: (LIST? OF STRING?)</code>	Returns a list containing the names of all graphs in the current composite database.
<code>graph.propertiesByName()</code>	<code>graph.propertiesByName(name :: STRING?) :: (MAP?)</code>	Returns a map containing the properties associated with the given graph.
<code>graph.byName()</code>	<code>USE graph.byName(name :: STRING?)</code>	Resolves a constituent graph by name.

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.	Using UDF	Extending Neo4j (UDF)
Aggregating	Consumes many rows and produces an aggregated result.	Using aggregating UDF	Extending Neo4j (Aggregating UDF)

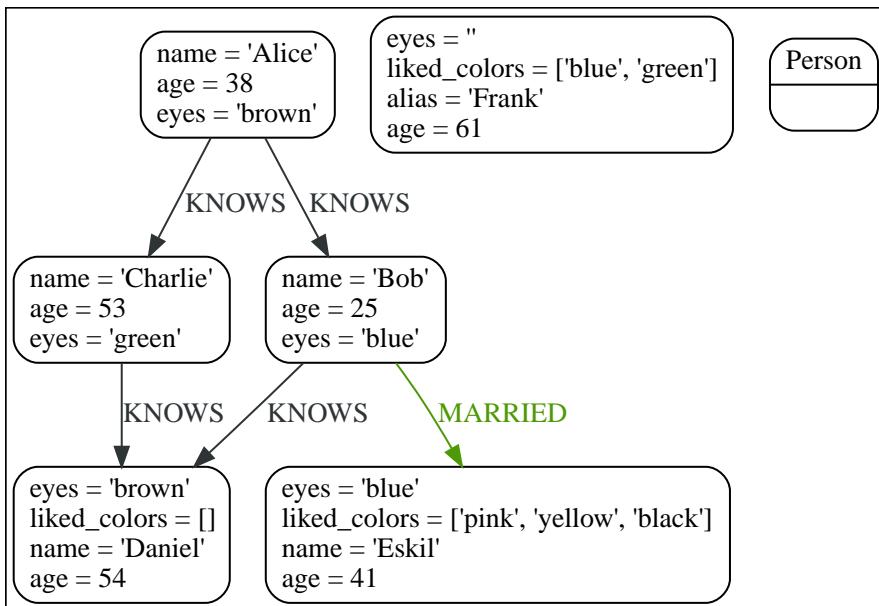
Predicate functions

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 if the predicate evaluates to `null` for at least one element and does not evaluate to `false` for any other element.

Syntax:

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

Returns:

```
A Boolean.
```

Arguments:

Name	Description
list	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.
variable	A variable that can be used from within the predicate.
predicate	A predicate that is tested against all items in the list.

Example 71. 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 295. 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 if the predicate evaluates to `null` for at least one element and does not evaluate to `true` for any other element.

Syntax:

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

Returns:

A Boolean.

Arguments:

Name	Description
list	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.
variable	A variable that can be used from within the predicate.
predicate	A predicate that is tested against all items in the list.

Example 72. 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 296. 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.

`null` is returned if the input argument is `null`.



To check if a property is not `null` use the `IS NOT NULL` predicate.

Syntax:

```
exists(pattern)
```

Returns:

A Boolean.

Arguments:

Name	Description
pattern	A pattern.

Example 73. 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 297. Result

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

Rows: 5



The function `exists()` looks very similar to the expression `EXISTS { ... }`, but they are not related.

See [Using EXISTS subqueries](#) for more information.

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
list	An expression that returns a list.

Example 74. 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 298. Result

n
Node[4]{eyes: "blue", liked_colors: ["pink", "yellow", "black"], name: "Eskil", age: 41}
Node[5]{eyes: "", liked_colors: ["blue", "green"], alias: "Frank", age: 61}
Rows: 2

Syntax:

```
isEmpty(map)
```

Returns:

A Boolean.

Arguments:

Name	Description
map	An expression that returns a map.

Example 75. isEmpty(map)

Query

```
MATCH (n)
WHERE isEmpty(properties(n))
RETURN n
```

Nodes that does not have any properties are returned.

Table 299. Result

n
Node[6]{}

Rows: 1

Syntax:

```
isEmpty(string)
```

Returns:

A Boolean.

Arguments:

Name	Description
string	An expression that returns a string.

Example 76. 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 300. 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 if the predicate evaluates to `null` for at least one element and does not evaluate to `true` for any other element.

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 77. 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 301. 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 if the predicate evaluates to `null` for at least one element and `true` for max one element.

Syntax:

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

Returns:

A Boolean.

Arguments:

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

Example 78. 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 302. Result

p
(0)-[KNOWS,0]->(1)
Rows: 1

Scalar functions

Scalar functions return a single value.

Functions:

- [coalesce\(\)](#)
- [elementId\(\)](#)
- [endNode\(\)](#)
- [head\(\)](#)
- [id\(\)](#) Deprecated
- [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.

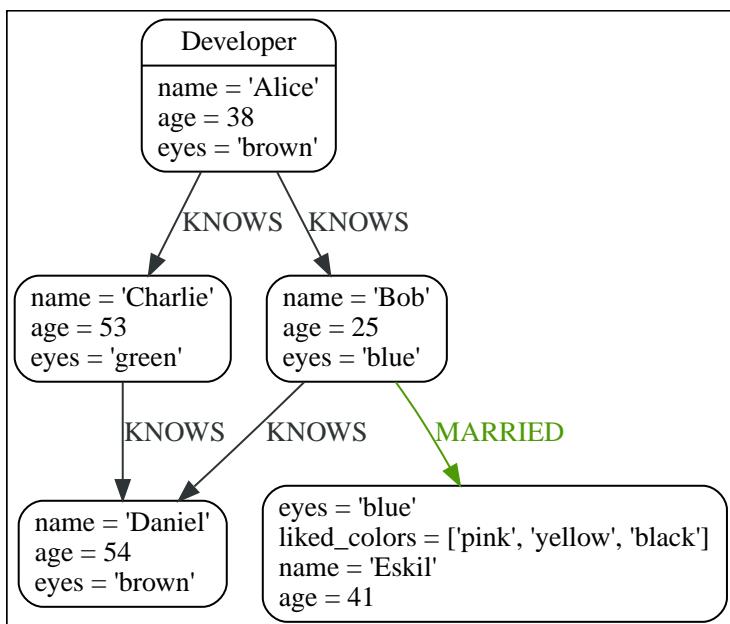


Function `length()`

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 79. coalesce()

Query

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

Table 303. Result

coalesce(a.hairColor, a.eyes)
"brown"
Rows: 1

elementId()

The function `elementId()` returns a node or relationship identifier, unique within a specific transaction and DBMS.



Every node and relationship is guaranteed an element ID. This ID is unique among both nodes and relationships across all databases in the same DBMS within the scope of a single transaction.

However, no guarantees are given regarding the order of the returned ID values or the length of the ID string values.

Outside of the scope of a single transaction, no guarantees are given about the mapping between ID values and elements.

Syntax:

elementId(expression)

Returns:

A String.

Arguments:

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

Considerations:

`elementId(null)` returns `null`.

`elementId` on values other than a node, relationship, or null will fail the query.

Example 80. `elementId()`

Query

```
MATCH (a)
RETURN elementId(a)
```

The node identifier for each of the nodes is returned.

Table 304. Result

elementId(a)
"4:c0a65d96-4993-4b0c-b036-e7ebd9174905:0"
"4:c0a65d96-4993-4b0c-b036-e7ebd9174905:1"
"4:c0a65d96-4993-4b0c-b036-e7ebd9174905:2"
"4:c0a65d96-4993-4b0c-b036-e7ebd9174905:3"
"4:c0a65d96-4993-4b0c-b036-e7ebd9174905:4"

Rows: 5

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 81. endNode()

Query

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

Table 305. 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
expression	An expression that returns a list.

Considerations:

`head(null)` returns `null`.

`head([])` returns `null`.

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

Example 82. 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 306. Result

a.liked_colors	head(a.liked_colors)
["pink", "yellow", "black"]	"pink"
Rows: 1	

id() Deprecated

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](#).



The function `id` is deprecated. Use the function `elementId` instead.



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 [composite database](#), the `id()` function should be used with caution. It is recommended to use `elementId()` instead.

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 83. id()

Query

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

The node identifier for each of the nodes is returned.

Table 307. 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 84. `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 308. Result

a.liked_colors	last(a.liked_colors)
["pink", "yellow", "black"]	"black"
Rows: 1	

length()

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

Syntax:

```
length(path)
```

Returns:

An Integer.

Arguments:

Name	Description
path	An expression that returns a path.

Considerations:

`length(null)` returns `null`.

Example 85. `length()`

Query

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

The length of the path `p` is returned.

Table 309. Result

length(p)
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
expression	An expression that returns a relationship, a node, or a map.

Considerations:

`properties(null)` returns `null`.

Example 86. `properties()`

Query

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

Table 310. Result

<code>properties(p)</code>
<code>{"city": "Berlin", "name": "Stefan"}</code>

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:

<code>randomUUID()</code>

Returns:

A String.

Example 87. randomUUID()

Query

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

Table 311. Result

uuid
"9f4c297d-309a-4743-a196-4525b96135c1"
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
list	An expression that returns a list.

Considerations:

`size(null)` returns `null`.

Example 88. size()

Query

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

Table 312. 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 comprehension)
```

Arguments:

Name	Description
pattern comprehension	A pattern comprehension that returns a list.

Example 89. size()

Query

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

Table 313. 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 90. `size()`

Query

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

Table 314. 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 91. `startNode()`

Query

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

Table 315. Result

<code>startNode(r)</code>
{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 92. timestamp()

Query

```
RETURN timestamp()
```

The time in milliseconds is returned.

Table 316. 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
expression	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 93. toBoolean()

Query

```
RETURN toBoolean('true'), toBoolean('not a boolean'), toBoolean(0)
```

Table 317. Result

toBoolean('true')	toBoolean('not a boolean')	toBoolean(0)
true	<null>	false
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 94. toBooleanOrNull()

Query

```
RETURN toBooleanOrNull('true'), toBooleanOrNull('not a boolean'), toBooleanOrNull(0),  
toBooleanOrNull(1.5)
```

Table 318. Result

toBooleanOrNull('true')	toBooleanOrNull('not a boolean')	toBooleanOrNull(0)	toBooleanOrNull(1.5)
true	<null>	false	<null>
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
expression	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 95. toFloat()

Query

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

Table 319. Result

toFloat('11.5')	toFloat('not a number')
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 96. toFloatOrNull()

Query

```
RETURN toFloatOrNull('11.5'), toFloatOrNull('not a number'), toFloatOrNull(true)
```

Table 320. Result

toFloatOrNull('11.5')	toFloatOrNull('not a number')	toFloatOrNull(true)
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
expression	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 97. tointeger()

Query

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

Table 321. Result

toInteger('42')	toInteger('not a number')	toInteger(true)
42	<null>	1
Rows: 1		

tolntegerOrNull()

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:

If `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 98. toIntegerOrNull()

Query

```
RETURN toIntegerOrNull('42'), toIntegerOrNull('not a number'), toIntegerOrNull(true),  
toIntegerOrNull(['A', 'B', 'C'])
```

Table 322. 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
relationship	An expression that returns a relationship.

Considerations:

```
type(null) returns null.
```

Example 99. type()

Query

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

The relationship type of `r` is returned.

Table 323. 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.

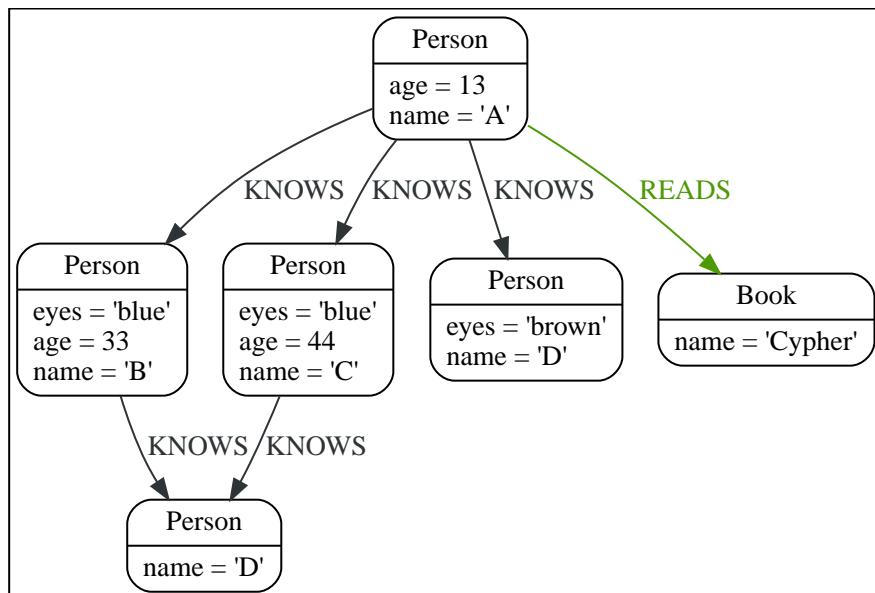
The input expression of an aggregation function can contain any expression, including expressions that are not grouping keys. However, not all expressions can be composed with aggregation functions. The example below will throw an error since we compose `n.x`, which is not a grouping key, with the aggregation expression `count(*)`. For more information see [Grouping keys](#).

```
RETURN n.x + count(*)
```

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 100. `avg()`

Query

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

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

Table 324. 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 101. avg()

Query

```
UNWIND [duration('P2DT3H'), duration('PT1H45S')] AS dur  
RETURN avg(dur)
```

The average of the two supplied Durations is returned.

Table 325. 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 102. collect()

Query

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

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

Table 326. 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.

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 103. `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 327. 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 104. `count()`

Query

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

The type of matched relationships are grouped and the group count are returned.

Table 328. 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 105. 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 329. 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 106. 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 330. 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 property types and lists thereof.

Considerations:

Any `null` values are excluded from the calculation.

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

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

`max(null)` returns `null`.

Example 107. `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 331. Result

max(val)
1
Rows: 1

Example 108. 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 332. Result

max(val)
[1, 2]
Rows: 1

Example 109. max()

Query

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

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

Table 333. Result

max(n.age)
44
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
expression	An expression returning a set containing any combination of property types and lists thereof.

Considerations:

Any `null` values are excluded from the calculation.

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

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

`min(null)` returns `null`.

Example 110. `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 334. Result

min(val)
"1"
Rows: 1

Example 111. `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 335. Result

min(val)
["a", "c", 23]
Rows: 1

Example 112. min()

Query

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

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

Table 336. 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 <code>null</code> values are excluded from the calculation.

<code>percentileCont(null, percentile)</code> returns <code>null</code> .

Example 113. 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 337. 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 114. percentileDisc()

Query

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

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

Table 338. 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
expression	A numeric expression.

Considerations:

Any <code>null</code> values are excluded from the calculation.

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

Example 115. 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 339. 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 116. 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 340. 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 117. sum()

Query

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

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

Table 341. 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
expression	An expression returning a set of Durations.

Considerations:

Any <code>null</code> values are excluded from the calculation.

Example 118. sum()

Query

```
UNWIND [duration('P2DT3H'), duration('PT1H45S')] AS dur
RETURN sum(dur)
```

The sum of the two supplied Durations is returned.

Table 342. Result

sum(dur)
P2DT4H45S
Rows: 1

Grouping keys

Aggregation expressions are expressions which contain one or more aggregation functions. A simple aggregation expression consists of a single aggregation function. For instance, `SUM(x.a)` is an aggregation expression that only consists of the aggregation function `SUM()` with `x.a` as its argument. Aggregation expressions are also allowed to be more complex, where the result of one or more aggregation functions are input arguments to other expressions. For instance, `0.1 * (SUM(x.a) / COUNT(x.b))` is an aggregation expression that contains two aggregation functions, `SUM()` with `x.a` as its argument and `COUNT()` with `x.b` as its argument. Both are input arguments to the division expression.

For aggregation expressions to be correctly computable for the buckets formed by the grouping key(s), they have to fulfill some requirements. Specifically, each sub expression in an aggregation expression has to be either:

- an aggregation function, e.g. `SUM(x.a)`,
- a constant, e.g. `0.1`,
- a parameter, e.g. `$param`,
- a grouping key, e.g. the `a` in `RETURN a, count(*)`
- a local variable, e.g. the `x` in `count(*) + size([x IN range(1, 10) | x])`, or
- a subexpression, all whose operands are operands allowed in an aggregation expression.

Examples of aggregation expressions.

Example 119. Simple aggregation without any grouping keys:

Query

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

Table 343. Result

max(p.age)
44
Rows: 1

Example 120. Addition of an aggregation and a constant, without any grouping keys:

Query

```
MATCH (p: Person) RETURN max(p.age) + 1
```

Table 344. Result

max(p.age) + 1
45
Rows: 1

Example 121. Subtraction of a property access and an aggregation.

Note that `n` is a grouping key:

Query

```
MATCH (n: Person{name: "A"})-[:KNOWS]-(f:Person) RETURN n, n.age - max(f.age)
```

Table 345. Result

n	n.age - max(f.age)
Node[0]{age:13,name: "A"}	-31
Rows: 1	

Example 122. Subtraction of a property access and an aggregation.

Note that `n.age` is a grouping key:

Query

```
MATCH (n: Person{name: "A"})-[:KNOWS]-(f:Person) RETURN n.age, n.age - max(f.age)
```

Table 346. Result

n.age	n.age - max(f.age)
13	-31
Rows: 1	

Grouping keys themselves can be complex expressions. For better query readability, Cypher only recognizes a sub-expression in aggregation expressions as a grouping key if the grouping key is either:

- A variable - e.g. the `n` in `RETURN n, n.age - max(f.age)`
- A property access - e.g. the `n.age` in `RETURN n.age, n.age - max(f.age)`
- A map access - e.g. the `n.age` in `WITH {age: 34, name:Chris} AS n RETURN n.age, n.age - max(n.age)`

If more complex grouping keys are needed as operands in aggregation expression, it is always possible to project them in advance with `WITH`.

Using the property `n.age` will throw an exception, since `n.age` is not a grouping key:

Query

```
MATCH (n: Person{name: "A"})-[:KNOWS]-(f:Person) RETURN n.age - max(f.age)
```

`n.age + n.age` is not a valid grouping key, since the expression is not a variable, property access or map access. It can therefore not be used in the expression which contains the aggregation function:

Query

```
MATCH (n: Person{name: "A"})-[:KNOWS]-(f:Person) RETURN n.age + n.age, n.age + n.age - max(f.age)
```

The above query could be rewritten to:

Query

```
MATCH (n: Person{name: "A"})-[:KNOWS]-(f:Person) WITH n.age + n.age AS groupingKey, f RETURN groupingKey, groupingKey - max(f.age)
```

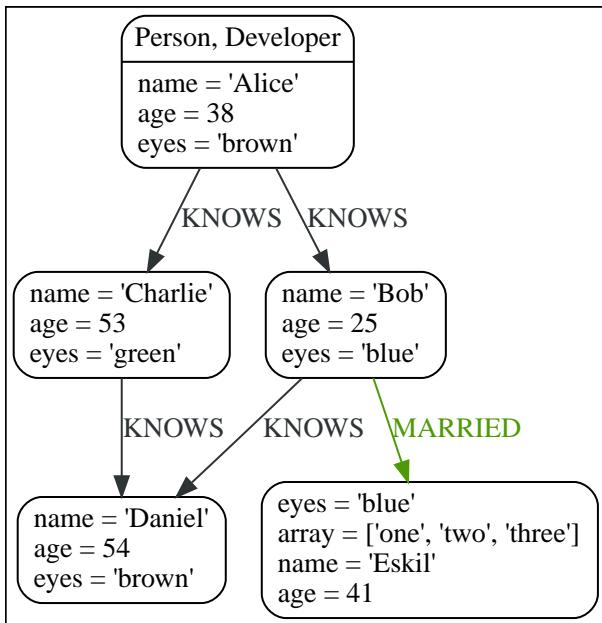
List functions

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

Considerations:

`keys(null)` returns `null`.

Example 123. 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 347. Result

keys(a)
<code>["name", "age", "eyes"]</code>
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
node	An expression that returns a single node.

Considerations:

`labels(null)` returns `null`.

Example 124. `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 348. Result

<code>labels(a)</code>
<code>["Person", "Developer"]</code>
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 125. 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 349. 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 126. 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 350. 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 127. 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 351. 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
path	An expression that returns a path.

Considerations:

```
relationships(null) returns null.
```

Example 128. 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 352. Result

relationships(p)
[:KNOWS[0]{}, :MARRIED[4]{})]
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 129. reverse()

Query

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

Table 353. 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 130. 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 354. 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 131. toBooleanList()

Query

```
RETURN toBooleanList(null) as noList,
       toBooleanList([null, null]) as nullsInList,
       toBooleanList(['a string', true, 'false', null, ['A','B']]) as mixedList
```

Table 355. 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 132. `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 356. Result

noList	nullsInList	mixedList
<null>	[<null>, <null>]	[<null>, 2.5, 3.14159, <null>, <null>]
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 133. `toIntegerList()`

Query

```
RETURN toIntegerList(null) as noList,
    toIntegerList([null, null]) as nullsInList,
    toIntegerList(['a string', 2, '5', null, ['A','B']]) as mixedList
```

Table 357. 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
list	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 134. `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 358. Result

noList	nullsInList	mixedList
<null>	[<null>,<null>]	["already a string", "2", "1955-11-05",<null>,<null>]
Rows: 1		

Mathematical functions - numeric

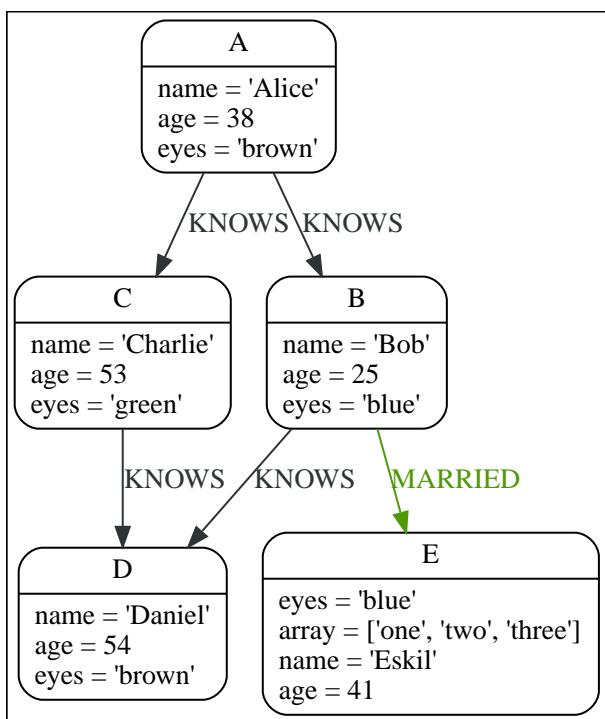
These functions all operate on numeric expressions only, and will return an error if used on any other values. See also [Mathematical operators](#).

Functions:

- [abs\(\)](#)
- [ceil\(\)](#)
- [floor\(\)](#)
- [isNaN\(\)](#)
- [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 135. 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 359. 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
expression	A numeric expression.

Considerations:

```
ceil(null) returns null.
```

Example 136. ceil()

Query

```
RETURN ceil(0.1)
```

The ceil of `0.1` is returned.

Table 360. 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
expression	A numeric expression.

Considerations:

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

Example 137. floor()

Query

```
RETURN floor(0.9)
```

The floor of `0.9` is returned.

Table 361. Result

<code>floor(0.9)</code>
<code>0.0</code>
Rows: 1

isNaN()

`isNaN()` returns `true` if the given numeric value is `NaN` (Not a Number).

Syntax:

```
isNaN(expression)
```

Returns:

A Boolean.

Arguments:

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

Considerations:

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

Example 138. isNaN()

Query

```
RETURN isNaN(0/0.0)
```

true is returned since the value is NaN.

Table 362. Result

isNaN(0/0.0)
true
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 139. rand()

Query

```
RETURN rand()
```

A random number is returned.

Table 363. 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
expression	A numeric expression to be rounded.

Considerations:

`round(null)` returns `null`.

Example 140. `round()`

Query

```
RETURN round(3.141592)
```

`3.0` is returned.

Table 364. Result

round(3.141592)
3.0
Rows: 1

Example 141. `round()` of negative number with tie

Query

```
RETURN round(-1.5)
```

Ties are rounded towards positive infinity, therefore `-1.0` is returned.

Table 365. 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 142. `round()` with precision

Query

```
RETURN round(3.141592, 3)
```

`3.142` is returned.

Table 366. Result

round(3.141592, 3)
3.142
Rows: 1

Example 143. round() with precision 0 and tie

Query

```
RETURN round(-1.5, 0)
```

To align with `round(-1.5)`, `-1.0` is returned.

Table 367. Result

round(-1.5, 0)
-1.0
Rows: 1

Example 144. 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 368. Result

round(-1.55, 1)
-1.6
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
expression	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 145. `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 369. Result

positive	negative	positiveTie	negativeTie
1.3	-1.3	1.3	-1.4

Rows: 1

Example 146. 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 370. Result

positive	negative	positiveTie	negativeTie
1.2	-1.2	1.2	-1.3
Rows: 1			

Example 147. 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 371. Result

positive	negative	positiveTie	negativeTie
1.3	-1.2	1.3	-1.3
Rows: 1			

Example 148. 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 372. Result

positive	negative	positiveTie	negativeTie
1.2	-1.3	1.2	-1.4
Rows: 1			

Example 149. 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 373. Result

positive	negative	positiveTie	negativeTie
1.2	-1.3	1.3	-1.4
Rows: 1			

Example 150. 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 374. Result

positive	negative	positiveTie	negativeTie
1.2	-1.3	1.2	-1.3
Rows: 1			

Example 151. 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 375. 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 152. `sign()`

Query

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

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

Table 376. 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 153. e()

Query

```
RETURN e()
```

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

Table 377. 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
expression	A numeric expression.

Considerations:

`exp(null)` returns `null`.

Example 154. `exp()`

Query

```
RETURN exp(2)
```

`e` to the power of `2` is returned.

Table 378. Result

<code>exp(2)</code>
7.38905609893065
Rows: 1

log()

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

Syntax:

```
log(expression)
```

Returns:

A Float.

Arguments:

Name	Description
expression	A numeric expression.

Considerations:

`log(null)` returns `null`.

`log(0)` returns `null`.

Example 155. log()

Query

```
RETURN log(27)
```

The natural logarithm of 27 is returned.

Table 379. 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
expression	A numeric expression.

Considerations:

`log10(null)` returns `null`.

`log10(0)` returns `null`.

Example 156. log10()

Query

```
RETURN log10(27)
```

The common logarithm of 27 is returned.

Table 380. 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
expression	A numeric expression.

Considerations:

<code>sqrt(null)</code> returns null.

<code>sqrt(<any negative number>)</code> returns NaN
--

Example 157. `sqrt()`

Query

```
RETURN sqrt(256)
```

The square root of 256 is returned.

Table 381. 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
expression	A numeric expression that represents the angle in radians.

Considerations:

`acos(null)` returns `null`.

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

Example 158. `acos()`

Query

```
RETURN acos(0.5)
```

The arccosine of `0.5` is returned.

Table 382. 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
expression	A numeric expression that represents the angle in radians.

Considerations:

`asin(null)` returns `null`.

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

Example 159. `asin()`

Query

```
RETURN asin(0.5)
```

The arcsine of `0.5` is returned.

Table 383. 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
expression	A numeric expression that represents the angle in radians.

Considerations:

`atan(null)` returns `null`.

Example 160. atan()

Query

```
RETURN atan(0.5)
```

The arctangent of `0.5` is returned.

Table 384. 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
expression1	A numeric expression for y that represents the angle in radians.
expression2	A numeric expression for x that represents the angle in radians.

Considerations:

<code>atan2(null, null)</code> , <code>atan2(null, expression2)</code> and <code>atan(expression1, null)</code> all return <code>null</code> .
--

Example 161. atan2()

Query

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

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

Table 385. 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
expression	A numeric expression that represents the angle in radians.

Considerations:

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

Example 162. cos()

Query

```
RETURN cos(0.5)
```

The cosine of `0.5` is returned.

Table 386. Result

<code>cos(0.5)</code>
<code>0.8775825618903728</code>
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:

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

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

Example 163. cot()

Query

```
RETURN cot(0.5)
```

The cotangent of `0.5` is returned.

Table 387. Result

cot(0.5)
1.830487721712452
Rows: 1

degrees()

`degrees()` converts radians to degrees.

Syntax:

```
degrees(expression)
```

Returns:

A Float.

Arguments:

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

Considerations:

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

Example 164. degrees

Query

```
RETURN degrees(3.14159)
```

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

Table 388. 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
expression	A numeric expression that represents the angle in radians.

Considerations:

<code>haversin(null)</code> returns <code>null</code> .

Example 165. haversin()

Query

```
RETURN haversin(0.5)
```

The haversine of 0.5 is returned.

Table 389. 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 166. 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 390. 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 167. pi()

Query

```
RETURN pi()
```

The constant pi is returned.

Table 391. 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 168. radians()

Query

```
RETURN radians(180)
```

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

Table 392. Result

radians(180)
3.141592653589793
Rows: 1

sin()

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

Syntax:

```
sin(expression)
```

Returns:

A Float.

Arguments:

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

Considerations:

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

Example 169. sin()

Query

```
RETURN sin(0.5)
```

The sine of `0.5` is returned.

Table 393. 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
expression	A numeric expression that represents the angle in radians.

Considerations:

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

Example 170. tan()

Query

```
RETURN tan(0.5)
```

The tangent of `0.5` is returned.

Table 394. Result

<code>tan(0.5)</code>
<code>0.5463024898437905</code>
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\(\)](#)
- [toLower\(\)](#)

- [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
original	An expression that returns a string.
length	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 171. `left()`

Query

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

Table 395. 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
original	An expression that returns a string.

Considerations:

```
ltrim(null) returns null.
```

Example 172. ltrim()

Query

```
RETURN ltrim('  hello')
```

Table 396. 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
original	An expression that returns a string.
search	An expression that specifies the string to be replaced in original.
replace	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 173. `replace()`

Query

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

Table 397. 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
original	An expression that returns a string.

Considerations:

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

Example 174. reverse

Query

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

Table 398. 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
original	An expression that returns a string.
length	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 <code>length</code> is not a positive integer, an error is raised.

If <code>length</code> exceeds the size of <code>original</code> , <code>original</code> is returned.

Example 175. right()

Query

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

Table 399. 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
original	An expression that returns a string.

Considerations:

```
rtrim(null) returns null.
```

Example 176. rtrim()

Query

```
RETURN rtrim('hello   ')
```

Table 400. 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 177. `split()`

Query

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

Table 401. 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
original	An expression that returns a string.
start	An expression that returns a positive integer, denoting the position at which the substring will begin.
length	An expression that returns a positive integer, denoting how many characters of original will be returned.

Considerations:

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

Example 178. `substring()`

Query

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

Table 402. Result

substring('hello', 1, 3)	substring('hello', 2)
"ell"	"llo"
Rows: 1	

`toLowerCase()`

`toLowerCase()` returns the original string in lowercase.

Syntax:

```
toLowerCase(original)
```

Returns:

A String.

Arguments:

Name	Description
original	An expression that returns a string.

Considerations:

`toLowerCase(null)` returns `null`.

Example 179. `toLowerCase()`

Query

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

Table 403. 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
expression	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 180. `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 404. 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 181. `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 405. 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- 11T12:31:14.34 1+01:00[Europe /Stockholm]"	"PT11M"	<null>

Rows: 1

`toUpperCase()`

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

Syntax:

```
toUpperCase(original)
```

Returns:

A String.

Arguments:

Name	Description
original	An expression that returns a string.

Considerations:

```
toUpperCase(null) returns null.
```

Example 182. toUpper()

Query

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

Table 406. 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
original	An expression that returns a string.

Considerations:

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

Example 183. trim()

Query

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

Table 407. 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 408. 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 be specified for the functions that retrieve the current instant. This allows for the retrieval of the current instant in the specified time zone.

Controlling which clock to use

The functions which create temporal instant values based on the current instant use the `statement` clock as default. However, there are three different clocks available for more fine-grained control:

- **transaction**: The same instant is produced for each invocation within the same transaction. A different time may be produced for different transactions.
- **statement**: The same instant is produced for each invocation within the same statement. A different time may be produced for different statements within the same transaction.
- **realtime**: The instant produced will be the live clock of the system.

The following table lists the different sub-functions for specifying the clock to be used when creating the current temporal instant value:

Type	default	transaction	statement	realtime
Date	<code>date()</code>	<code>date.transaction()</code>	<code>date.statement()</code>	<code>date.realtime()</code>
Time	<code>time()</code>	<code>time.transaction()</code>	<code>time.statement()</code>	<code>time.realtime()</code>
LocalTime	<code>localtime()</code>	<code>localtime.transaction()</code>	<code>localtime.statement()</code>	<code>localtime.realtime()</code>
DateTime	<code>datetime()</code>	<code>datetime.transaction()</code>	<code>datetime.statement()</code>	<code>datetime.realtime()</code>
LocalDateTime	<code>localdatetime()</code>	<code>localdatetime.transaction()</code>	<code>localdatetime.statement()</code>	<code>localdatetime.realtime()</code>

Truncating temporal values

A temporal instant value can be created by truncating another temporal instant value at the nearest preceding point in time at a specified component boundary (namely, a truncation unit). A temporal instant value created in this way will have all components which are less significant than the specified truncation unit set to their default values.

It is possible to supplement the truncated value by providing a map containing components which are less significant than the truncation unit. This will have the effect of overriding the default values which would otherwise have been set for these less significant components.

The following truncation units are supported:

- **millennium**: Select the temporal instant corresponding to the 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	date.truncate('mille nnium', input)			datetime.truncate(' millennium', input)	localdatetime.trunc ate('millennium', input)
century	date.truncate('cent ury', input)			datetime.truncate(' century', input)	localdatetime.trunc ate('century', input)
decade	date.truncate('deca de', input)			datetime.truncate(' decade', input)	localdatetime.trunc ate('decade', input)
year	date.truncate('year' , input)			datetime.truncate(' year', input)	localdatetime.trunc ate('year', input)
weekYear	date.truncate('wee kYear', input)			datetime.truncate(' weekYear', input)	localdatetime.trunc ate('weekYear', input)
quarter	date.truncate('qua rter', input)			datetime.truncate(' quarter', input)	localdatetime.trunc ate('quarter', input)
month	date.truncate('mon th', input)			datetime.truncate(' month', input)	localdatetime.trunc ate('month', input)
week	date.truncate('wee k', input)			datetime.truncate(' week', input)	localdatetime.trunc ate('week', input)
day	date.truncate('day', input)	time.truncate('day', input)	localtime.truncate(' day', input)	datetime.truncate(' day', input)	localdatetime.trunc ate('day', input)
hour		time.truncate('hour' , input)	localtime.truncate(' hour', input)	datetime.truncate(' hour', input)	localdatetime.trunc ate('hour', input)
minute		time.truncate('minu te', input)	localtime.truncate(' minute', input)	datetime.truncate(' minute', input)	localdatetime.trunc ate('minute', input)
second		time.truncate('seco nd', input)	localtime.truncate(' second', input)	datetime.truncate(' second', input)	localdatetime.trunc ate('second', input)
millisecond		time.truncate('millis econd', input)	localtime.truncate(' millisecond', input)	datetime.truncate(' millisecond', input)	localdatetime.trunc ate('millisecond', input)
microsecond		time.truncate('micr osecond', input)	localtime.truncate(' microsecond', input)	datetime.truncate(' microsecond', input)	localdatetime.trunc ate('microsecond', input)

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 time zone .

Considerations:

If no parameters are provided, `date()` must be invoked (`date({})` is invalid).

Example 184. `date()`

Query

```
RETURN date() AS currentDate
```

The current date is returned.

Table 409. Result

currentDate
2022-06-14
Rows: 1

Example 185. date()

Query

```
RETURN date({timezone: 'America/Los Angeles'}) AS currentDateInLA
```

The current date in California is returned.

Table 410. 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
timezone	A string expression that represents the <code>time zone</code> .

Example 186. date.transaction()

Query

```
RETURN date.transaction() AS currentDate
```

Table 411. 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 time zone .

Example 187. `date.statement()`

Query

```
RETURN date.statement() AS currentDate
```

Table 412. 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 time zone .

Example 188. date.realtime()

Query

```
RETURN date.realtime() AS currentDate
```

Table 413. Result

currentDate
2022-06-14
Rows: 1

Example 189. date.realtime()

Query

```
RETURN date.realtime('America/Los Angeles') AS currentDateInLA
```

Table 414. 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 least four digits 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 190. date()

Query

```
UNWIND [
  date({year: 1984, month: 10, day: 11}),
  date({year: 1984, month: 10}),
  date({year: 1984})
] AS theDate
RETURN theDate
```

Table 415. 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 191. date()

Query

```
UNWIND [
  date({year: 1984, week: 10, dayOfWeek: 3}),
  date({year: 1984, week: 10}),
  date({year: 1984})
] AS theDate
RETURN theDate
```

Table 416. 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:	
year	An expression consisting of at least four digits that specifies the year.
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.

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 192. `date()`

Query

```
UNWIND [
  date({year: 1984, quarter: 3, dayOfQuarter: 45}),
  date({year: 1984, quarter: 3}),
  date({year: 1984})
] AS theDate
RETURN theDate
```

Table 417. 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 193. `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 418. 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
<code>temporalValue</code>	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 194. `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 419. 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 195. `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 420. 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 196. 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 421. Result

truncMillenium	truncCentury	truncDecade	truncYear	truncWeekYear	truncQuarter	truncMonth	truncWeek	truncDay
2000-01-01	2000-01-01	2010-01-01	2017-01-05	2017-01-02	2017-10-01	2017-11-01	2017-11-07	2017-11-11
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:	
timezone	A string expression that represents the time zone .

Considerations:

If no parameters are provided, `datetime()` must be invoked (`datetime({})` is invalid).

Example 197. .datetime()

Query

```
RETURN datetime() AS currentDateTime
```

The current date and time using the local time zone is returned.

Table 422. Result

currentDateTime
2022-06-14T10:02:28.192Z
Rows: 1

Example 198. .datetime()

Query

```
RETURN datetime({timezone: 'America/Los Angeles'}) AS currentDateTimeInLA
```

The current date and time of day in California is returned.

Table 423. 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
timezone	A string expression that represents the <code>time zone</code> .

Example 199. datetime.transaction()

Query

```
RETURN datetime.transaction() AS currentDateTime
```

Table 424. Result

currentDateTime
2022-06-14T10:02:28.290Z
Rows: 1

Example 200. datetime.transaction()

Query

```
RETURN datetime.transaction('America/Los Angeles') AS currentDateTimeInLA
```

Table 425. 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 <code>DateTime</code> .

Arguments:

Name	Description
<code>timezone</code>	A string expression that represents the time zone .

Example 201. datetime.statement()

Query

```
RETURN datetime.statement() AS currentDateTime
```

Table 426. 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 time zone .

Example 202. `datetime.realtime()`

Query

```
RETURN datetime.realtime() AS currentDateTime
```

Table 427. 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 `month` is omitted.

The day of the month component will default to 1 if `day` is omitted.

The hour component will default to 0 if `hour` is omitted.

The minute component will default to 0 if `minute` is omitted.

The second component will default to 0 if `second` is omitted.

Any missing `millisecond`, `microsecond` or `nanosecond` values will default to 0.

The `timezone` component will default to the configured default time zone if `timezone` is omitted.

If `millisecond`, `microsecond` and `nanosecond` are given in combination (as part of the same set of parameters), the individual values must be in the range 0 to 999.

The least significant components in the set `year`, `month`, `day`, `hour`, `minute`, and `second` may be omitted; i.e. it is possible to specify only `year`, `month` and `day`, but specifying `year`, `month`, `day` and `minute` is not permitted.

One or more of `millisecond`, `microsecond` and `nanosecond` can only be specified as long as `second` is also specified.

Example 203. 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 428. 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
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.
hour	An integer between 0 and 23 that specifies the hour of the day.
minute	An integer between 0 and 59 that specifies the number of minutes.
second	An integer between 0 and 59 that specifies the number of seconds.
millisecond	An integer between 0 and 999 that specifies the number of milliseconds.
microsecond	An integer between 0 and 999,999 that specifies the number of microseconds.
nanosecond	An integer between 0 and 999,999,999 that specifies the number of nanoseconds.
timezone	An expression that specifies the time zone.

Considerations:

The week component will default to 1 if `week` is omitted.

The day of the week component will default to 1 if `dayOfWeek` is omitted.

The hour component will default to 0 if `hour` is omitted.

The minute component will default to 0 if `minute` is omitted.

The second component will default to 0 if `second` is omitted.

Any missing `millisecond`, `microsecond` or `nanosecond` values will default to 0.

The timezone component will default to the configured default time zone if `timezone` is omitted.

If `millisecond`, `microsecond` and `nanosecond` are given in combination (as part of the same set of parameters), the individual values must be in the range 0 to 999.

The least significant components in the set `year`, `week`, `dayOfWeek`, `hour`, `minute`, and `second` may be omitted; i.e. it is possible to specify only `year`, `week` and `dayOfWeek`, but specifying `year`, `week`, `dayOfWeek` and `minute` is not permitted.

One or more of `millisecond`, `microsecond` and `nanosecond` can only be specified as long as `second` is also specified.

Example 204. 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 429. 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
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.
hour	An integer between 0 and 23 that specifies the hour of the day.
minute	An integer between 0 and 59 that specifies the number of minutes.
second	An integer between 0 and 59 that specifies the number of seconds.
millisecond	An integer between 0 and 999 that specifies the number of milliseconds.
microsecond	An integer between 0 and 999,999 that specifies the number of microseconds.
nanosecond	An integer between 0 and 999,999,999 that specifies the number of nanoseconds.
timezone	An expression that specifies the time zone.

Considerations:

The 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 205. 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 430. 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
second	An integer between <code>0</code> and <code>59</code> that specifies the number of seconds.
millisecond	An integer between <code>0</code> and <code>999</code> that specifies the number of milliseconds.
microsecond	An integer between <code>0</code> and <code>999,999</code> that specifies the number of microseconds.
nanosecond	An integer between <code>0</code> and <code>999,999,999</code> that specifies the number of nanoseconds.
timezone	An expression that specifies the time zone.

Considerations:

The ordinal day of the year component will default to <code>1</code> if <code>ordinalDay</code> is omitted.
The hour component will default to <code>0</code> if <code>hour</code> is omitted.
The minute component will default to <code>0</code> if <code>minute</code> is omitted.
The second component will default to <code>0</code> if <code>second</code> is omitted.
Any missing <code>millisecond</code> , <code>microsecond</code> or <code>nanosecond</code> values will default to <code>0</code> .
The timezone 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 <code>0</code> to <code>999</code> .
The least significant components in the set <code>year</code> , <code>ordinalDay</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> and <code>ordinalDay</code> , but specifying <code>year</code> , <code>ordinalDay</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 206. 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 431. 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
temporalValue	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 207. 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 432. 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 DateTime value.
<code>date</code>	A Date value.
<code>time</code>	A Time 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 208. `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 dateHHMMSSTimezone,
    datetime({date: dd, day: 28, hour: 10, minute: 10, second: 10}) AS dateDDHHMMSS,
    datetime({date: dd, day: 28, hour: 10, minute: 10, second: 10, timezone: 'Pacific/Honolulu'}) AS
dateDDHHMMSSTimezone
```

Table 433. Result

dateHHMMSS	dateHMHMSSTimezone	dateDDHHMMSS	dateDDHHMMSSTimezone
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 209. `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 434. 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 210. 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
dateTimeDDSTimezone
```

Table 435. Result

dateTime	dateTimeTimezone	dateTimeDDSS	dateTimeDDSTimezone
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 211. 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
dateTimeDDSTimezone
```

Table 436. Result

dateTime	dateTimeTimezone	dateTimeDDSS	dateTimeDDSTimezone
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 212. `datetime()`

Query

```
RETURN datetime({epochSeconds: timestamp() / 1000, nanosecond: 23}) AS theDate
```

Table 437. Result

theDate
2022-06-14T10:02:30.00000023Z
Rows: 1

Example 213. datetime()

Query

```
RETURN datetime({epochMillis: 424797300000}) AS theDate
```

Table 438. 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: <code>DateTime</code> , <code>LocalDateTime</code> , <code>Date</code> .

Name	Description
mapOfComponents	An expression evaluating to a map containing components less significant than <code>unit</code> . During truncation, a time zone can be attached or overridden using the key <code>timezone</code> .

Considerations:

<code>temporalInstantValue</code> cannot be a Date value if <code>unit</code> is one of: 'hour', 'minute', 'second', 'millisecond', 'microsecond'.
The time zone of <code>temporalInstantValue</code> may be overridden; for example, <code>datetime.truncate('minute', input, {timezone: '+0200'})</code> .
If <code>temporalInstantValue</code> is one of Time, DateTime — a value with a time zone — and the time zone is overridden, no time conversion occurs.
If <code>temporalInstantValue</code> 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 <code>mapOfComponents</code> must be less significant than <code>unit</code> ; i.e. if <code>unit</code> is 'day', <code>mapOfComponents</code> cannot contain information pertaining to a month.
Any component that is not contained in <code>mapOfComponents</code> and which is less significant than <code>unit</code> will be set to its minimal value.
If <code>mapOfComponents</code> is not provided, all components of the returned value which are less significant than <code>unit</code> will be set to their default values.
If <code>temporalInstantValue</code> is not provided, it will be set to the current date, time and timezone, i.e. <code>datetime.truncate(unit)</code> is equivalent of <code>datetime.truncate(unit, datetime())</code> .

Example 214. `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 439. 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 time zone .

Considerations:

If no parameters are provided, `localdatetime()` must be invoked (`localdatetime({})` is invalid).

Example 215. localdatetime()

Query

```
RETURN localdatetime() AS now
```

The current local date and time (i.e. in the local time zone) is returned.

Table 440. Result

now
2022-06-14T10:02:30.447
Rows: 1

Example 216. localdatetime()

Query

```
RETURN localdatetime({{timezone: 'America/Los Angeles'}}) AS now
```

The current local date and time in California is returned.

Table 441. 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
timezone	A string expression that represents the time zone .

Example 217. localdatetime.transaction()

Query

```
RETURN localdatetime.transaction() AS now
```

Table 442. Result

now
2022-06-14T10:02:30.532
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
timezone	A string expression that represents the time zone .

Example 218. localdatetime.statement()

Query

```
RETURN localdatetime.statement() AS now
```

Table 443. 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 time zone .

Example 219. localdatetime.realtime()

Query

```
RETURN localdatetime.realtime() AS now
```

Table 444. Result

now
2022-06-14T10:02:30.647817
Rows: 1

Example 220. localdatetime.realtime()

Query

```
RETURN localdatetime.realtime('America/Los Angeles') AS nowInLA
```

Table 445. 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 <code>LocalDateTime</code> .

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
nanosecond	An integer between 0 and 999,999,999 that specifies the number of nanoseconds.

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.
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 221. 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 446. 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:	
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.
hour	An integer between 0 and 23 that specifies the hour of the day.
minute	An integer between 0 and 59 that specifies the number of minutes.
second	An integer between 0 and 59 that specifies the number of seconds.
millisecond	An integer between 0 and 999 that specifies the number of milliseconds.
microsecond	An integer between 0 and 999,999 that specifies the number of microseconds.
nanosecond	An integer between 0 and 999,999,999 that specifies the number of nanoseconds.

Considerations:

The 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 222. localdatetime()

Query

```
RETURN
localdatetime({
    year: 1984, week: 10, dayOfWeek: 3,
    hour: 12, minute: 31, second: 14, millisecond: 645
}) AS theDate
```

Table 447. 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 <code>1</code> and <code>4</code> that specifies the quarter.
<code>dayOfQuarter</code>	An integer between <code>1</code> and <code>92</code> that specifies the day of the quarter.
<code>hour</code>	An integer between <code>0</code> and <code>23</code> that specifies the hour of the day.
<code>minute</code>	An integer between <code>0</code> and <code>59</code> that specifies the number of minutes.
<code>second</code>	An integer between <code>0</code> and <code>59</code> that specifies the number of seconds.

Name	Description
millisecond	An integer between 0 and 999 that specifies the number of milliseconds.
microsecond	An integer between 0 and 999,999 that specifies the number of microseconds.
nanosecond	An integer between 0 and 999,999,999 that specifies the number of nanoseconds.

Considerations:

The quarter component will default to 1 if <code>quarter</code> is omitted.
The day of the quarter component will default to 1 if <code>dayOfQuarter</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.
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>quarter</code> , <code>dayOfQuarter</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>quarter</code> and <code>dayOfQuarter</code> , but specifying <code>year</code> , <code>quarter</code> , <code>dayOfQuarter</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 223. localdatetime()

Query

```
RETURN
localdatetime({
    year: 1984, quarter: 3, dayOfQuarter: 45,
    hour: 12, minute: 31, second: 14, nanosecond: 645876123
}) AS theDate
```

Table 448. 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 224. localdatetime()

Query

```
RETURN
localdatetime({
    year: 1984, ordinalDay: 202,
    hour: 12, minute: 31, second: 14, microsecond: 645876
}) AS theDate
```

Table 449. 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 <code>LocalDateTime</code> .

Arguments:

Name	Description
<code>temporalValue</code>	A string representing a temporal value.

Considerations:

<code>temporalValue</code> must comply with the format defined for <code>dates</code> and <code>times</code> .
--

<code>temporalValue</code> must denote a valid date and time; i.e. a <code>temporalValue</code> denoting <code>30 February 2001</code> is invalid.
--

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

Example 225. localdatetime()

Query

```
UNWIND [
localdatetime('2015-07-21T21:40:32.142'),
localdatetime('2015-W30-2T214032.142'),
localdatetime('2015-202T21:40:32'),
localdatetime('2015202T21')
] AS theDate
RETURN theDate
```

Table 450. 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]}) | localdatetime({date [, year, ..., nanosecond]}) |
localdatetime({time [, year, ..., nanosecond]}) | localdatetime({date, time [, year, ..., nanosecond]})
```

Returns:

A `LocalDateTime`.

Arguments:

Name	Description
A single map consisting of the following:	
<code>datetime</code>	A <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
day	An integer between 1 and 31 that specifies the day of the month.
week	An integer between 1 and 53 that specifies the week.
dayOfWeek	An integer between 1 and 7 that specifies the day of the week.
quarter	An integer between 1 and 4 that specifies the quarter.
dayOfQuarter	An integer between 1 and 92 that specifies the day of the quarter.
ordinalDay	An integer between 1 and 366 that specifies the ordinal day of the year.
hour	An integer between 0 and 23 that specifies the hour of the day.
minute	An integer between 0 and 59 that specifies the number of minutes.
second	An integer between 0 and 59 that specifies the number of seconds.
millisecond	An integer between 0 and 999 that specifies the number of milliseconds.
microsecond	An integer between 0 and 999,999 that specifies the number of microseconds.
nanosecond	An integer between 0 and 999,999,999 that specifies the number of nanoseconds.

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 226. 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 451. Result

dateHHMMSS	dateDDHHMMSS
1984-10-11T10:10:10	1984-10-28T10:10:10
Rows: 1	

Example 227. 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 452. Result

YYYYMMDDTime	YYYYMMDDTimeSS
1984-10-11T12:31:14.645876	1984-10-11T12:31:42.645876
Rows: 1	

Example 228. 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 453. Result

dateTime	dateTimeDDSS
1984-10-11T12:31:14.645876	1984-10-28T12:31:42.645876
Rows: 1	

Example 229. 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 454. 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 230. 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 455. Result

truncMillenium	truncYear	truncMonth	truncDay	truncHour	truncSecond
2000-01-01T00:00	2017-01-02T00:00	2017-11-01T00:00	2017-11-11T00:00	2017-11-11T12:00:00.000000002	2017-11-11T12:31:14

Rows: 1

localtime()

Details for using the `localtime()` function.

- Getting the current LocalTime
 - `localtime.transaction()`
 - `localtime.statement()`
 - `localtime.realpath()`
- 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 time zone .

Considerations:

If no parameters are provided, `localtime()` must be invoked (`localtime({})` is invalid).

Example 231. `localtime()`

Query

```
RETURN localtime() AS now
```

The current local time (i.e. in the local time zone) is returned.

Table 456. Result

now
10:02:31.596
Rows: 1

Example 232. `localtime()`

Query

```
RETURN localtime({timezone: 'America/Los Angeles'}) AS nowInLA
```

The current local time in California is returned.

Table 457. 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
timezone	A string expression that represents the time zone .

Example 233. localtime.transaction()

Query

```
RETURN localtime.transaction() AS now
```

Table 458. 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
timezone	A string expression that represents the time zone .

Example 234. localtime.statement()

Query

```
RETURN localtime.statement() AS now
```

Table 459. Result

now

10:02:31.697

Rows: 1

Example 235. localtime.statement()

Query

```
RETURN localtime.statement('America/Los Angeles') AS nowInLA
```

Table 460. 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 time zone .

Example 236. localtime.realtime()

Query

```
RETURN localtime.realtime() AS now
```

Table 461. 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:	
hour	An integer between 0 and 23 that specifies the hour of the day.
minute	An integer between 0 and 59 that specifies the number of minutes.
second	An integer between 0 and 59 that specifies the number of seconds.
millisecond	An integer between 0 and 999 that specifies the number of milliseconds.
microsecond	An integer between 0 and 999,999 that specifies the number of microseconds.
nanosecond	An integer between 0 and 999,999,999 that specifies the number of nanoseconds.

Considerations:

The 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 237. 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 462. 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 238. `localtime()`

Query

```
UNWIND [
  localtime('21:40:32.142'),
  localtime('214032.142'),
  localtime('21:40'),
  localtime('21')
] AS theTime
RETURN theTime
```

Table 463. 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 <code>0</code> and <code>23</code> that specifies the hour of the day.

Name	Description
minute	An integer between <code>0</code> and <code>59</code> that specifies the number of minutes.
second	An integer between <code>0</code> and <code>59</code> that specifies the number of seconds.
millisecond	An integer between <code>0</code> and <code>999</code> that specifies the number of milliseconds.
microsecond	An integer between <code>0</code> and <code>999,999</code> that specifies the number of microseconds.
nanosecond	An integer between <code>0</code> and <code>999,999,999</code> that specifies the number of nanoseconds.

Considerations:

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 239. `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 464. 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
unit	A string expression evaluating to one of the following strings: 'day', 'hour', 'minute', 'second', 'millisecond', 'microsecond'.
temporalInstantValue	An expression of one of the following types: DateTime, LocalDateTime, Time, LocalTime.
mapOfComponents	An expression evaluating to a map containing components less significant than unit.

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 240. 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 465. 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 time zone.

Considerations:

If no parameters are provided, `time()` must be invoked (`time({})` is invalid).

Example 241. `time()`

Query

```
RETURN time() AS currentTime
```

The current time of day using the local time zone is returned.

Table 466. Result

currentTime
10:02:32.192Z
Rows: 1

Example 242. `time()`

Query

```
RETURN time({timezone: 'America/Los Angeles'}) AS currentTimeInLA
```

The current time of day in California is returned.

Table 467. 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
timezone	A string expression that represents the time zone .

Example 243. time.transaction()

Query

```
RETURN time.transaction() AS currentTime
```

Table 468. Result

currentTime
10:02:32.276Z
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
timezone	A string expression that represents the time zone .

Example 244. time.statement()

Query

```
RETURN time.statement() AS currentTime
```

Table 469. Result

currentTime
10:02:32.317Z
Rows: 1

Example 245. time.statement()

Query

```
RETURN time.statement('America/Los Angeles') AS currentTimeInLA
```

Table 470. Result

currentTimeInLA
03:02:32.351-07:00
Rows: 1

time.realpath()

`time.realpath()` returns the current *Time* value using the `realtime` clock. This value will be the live clock of the system.

Syntax:

```
time.realpath([{timezone}])
```

Returns:

A Time.

Arguments:

Name	Description
timezone	A string expression that represents the time zone .

Example 246. time.realpath()

Query

```
RETURN time.realpath() AS currentTime
```

Table 471. 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:	
hour	An integer between 0 and 23 that specifies the hour of the day.
minute	An integer between 0 and 59 that specifies the number of minutes.
second	An integer between 0 and 59 that specifies the number of seconds.
millisecond	An integer between 0 and 999 that specifies the number of milliseconds.
microsecond	An integer between 0 and 999,999 that specifies the number of microseconds.
nanosecond	An integer between 0 and 999,999,999 that specifies the number of nanoseconds.
timezone	An expression that specifies the time zone.

Considerations:

The hour component will default to <code>0</code> if <code>hour</code> is omitted.
The minute component will default to <code>0</code> if <code>minute</code> is omitted.
The second component will default to <code>0</code> if <code>second</code> is omitted.
Any missing <code>millisecond</code> , <code>microsecond</code> or <code>nanosecond</code> values will default to <code>0</code> .
The timezone 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 <code>0</code> to <code>999</code> .
The least significant components in the set <code>hour</code> , <code>minute</code> , and <code>second</code> may be omitted; i.e. it is possible to specify only <code>hour</code> and <code>minute</code> , but specifying <code>hour</code> and <code>second</code> is not permitted.
One or more of <code>millisecond</code> , <code>microsecond</code> and <code>nanosecond</code> can only be specified as long as <code>second</code> is also specified.

Example 247. `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 472. 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
temporalValue	A string representing a temporal value.

Considerations:

<code>temporalValue</code> 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.
<code>temporalValue</code> must denote a valid time; i.e. a <code>temporalValue</code> denoting <code>15:67</code> is invalid.
<code>time(null)</code> returns <code>null</code> .

Example 248. `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 473. Result

theTime
21:40:32.142+01:00
21:40:32.142Z
21:40:32+01:00
21:40:32-01:00
21:40-01:30
21:40Z
21:40-02:00
22:00+18:00
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 Time value.
<code>hour</code>	An integer between <code>0</code> and <code>23</code> that specifies the hour of the day.
<code>minute</code>	An integer between <code>0</code> and <code>59</code> that specifies the number of minutes.
<code>second</code>	An integer between <code>0</code> and <code>59</code> that specifies the number of seconds.
<code>millisecond</code>	An integer between <code>0</code> and <code>999</code> that specifies the number of milliseconds.
<code>microsecond</code>	An integer between <code>0</code> and <code>999,999</code> that specifies the number of microseconds.
<code>nanosecond</code>	An integer between <code>0</code> and <code>999,999,999</code> that specifies the number of nanoseconds.
<code>timezone</code>	An expression that specifies the time zone.

Considerations:

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 249. 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 474. 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
mapOfComponents	An expression evaluating to a map containing components less significant than <code>unit</code> . During truncation, a time zone can be attached or overridden using the key <code>timezone</code> .

Considerations:

Truncating time to day — i.e. <code>unit</code> is <code>'day'</code> — is supported, and yields midnight at the start of the day (<code>00:00</code>), regardless of the value of <code>temporalInstantValue</code> . However, the time zone of <code>temporalInstantValue</code> is retained.
The time zone of <code>temporalInstantValue</code> may be overridden; for example, <code>time.truncate('minute', input, {timezone: '+0200'})</code> .
If <code>temporalInstantValue</code> is one of <code>Time</code> , <code>DateTime</code> — a value with a time zone — and the time zone is overridden, no time conversion occurs.
If <code>temporalInstantValue</code> is one of <code>LocalTime</code> , <code>LocalDateTime</code> , <code>Date</code> — 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 <code>mapOfComponents</code> must be less significant than <code>unit</code> ; i.e. if <code>unit</code> is <code>'second'</code> , <code>mapOfComponents</code> cannot contain information pertaining to a minute.
Any component that is not contained in <code>mapOfComponents</code> and which is less significant than <code>unit</code> will be set to its <code>minimal value</code> .
If <code>mapOfComponents</code> is not provided, all components of the returned value which are less significant than <code>unit</code> will be set to their default values.
If <code>temporalInstantValue</code> is not provided, it will be set to the current time and timezone, i.e. <code>time.truncate(unit)</code> is equivalent of <code>time.truncate(unit, time())</code> .

Example 250. `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 475. Result

truncDay	truncHour	truncMinute	truncSecond	truncMillisecond	truncMicrosecond
00:00-01:00	12:00-01:00	12:31-01:00	12:31:14-01:00	12:31:14.6450000	12:31:14.645876-01:00
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:	
<code>years</code>	A numeric expression.
<code>quarters</code>	A numeric expression.
<code>months</code>	A numeric expression.
<code>weeks</code>	A numeric expression.
<code>days</code>	A numeric expression.
<code>hours</code>	A numeric expression.
<code>minutes</code>	A numeric expression.
<code>seconds</code>	A numeric expression.
<code>milliseconds</code>	A numeric expression.
<code>microseconds</code>	A numeric expression.
<code>nanoseconds</code>	A numeric expression.

Considerations:

At least one parameter must be provided (`duration()` and `duration({})` are invalid).

There is no constraint on how many of the parameters are provided.

It is possible to have a Duration where the amount of a smaller unit (e.g. `seconds`) exceeds the threshold of a larger unit (e.g. `days`).

The values of the parameters may be expressed as decimal fractions.

The values of the parameters may be arbitrarily large.

The values of the parameters may be negative.

Example 251. 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 476. 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
temporalAmount	A string representing a temporal amount.

Considerations:

`temporalAmount` must comply with either the [unit based form](#) or [date-and-time based form](#) defined for [Durations](#).

Example 252. 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 477. 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
instant1	An expression returning any temporal instant type (Date etc) that represents the starting instant.
instant2	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.

Example 253. 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 478. 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
instant1	An expression returning any temporal instant type (Date etc) that represents the starting instant.
instant2	An expression returning any temporal instant type (Date etc) that represents the ending instant.

Considerations:

If <code>instant2</code> occurs earlier than <code>instant1</code> , the resulting Duration will be negative.
If <code>instant1</code> has a time component and <code>instant2</code> does not, the time component of <code>instant2</code> is assumed to be midnight, and vice versa.
If <code>instant1</code> has a time zone component and <code>instant2</code> does not, the time zone component of <code>instant2</code> is assumed to be the same as that of <code>instant1</code> , and vice versa.
If <code>instant1</code> has a date component and <code>instant2</code> does not, the date component of <code>instant2</code> is assumed to be the same as that of <code>instant1</code> , and vice versa.
Any difference smaller than a whole month is disregarded.

Example 254. 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 479. 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
instant1	An expression returning any temporal instant type (Date etc) that represents the starting instant.
instant2	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 255. 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 480. 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 (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.

Example 256. 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(date("2015-06-24"), localtime("14:30")),
duration.inSeconds(datetime({year: 2017, month: 10, day: 29, hour: 0, timezone: 'Europe/Stockholm'})),
datetime({year: 2017, month: 10, day: 29, hour: 0, timezone: 'Europe/London'}))
] AS aDuration
RETURN aDuration
```

Table 481. Result

aDuration
PT24H
PT-24H
PT25H32.142S
PT14H30M
PT1H
Rows: 5

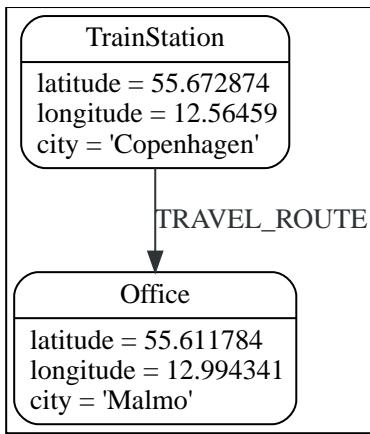
Spatial functions

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 257. 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 482. Result

dist
1.5
Rows: 1

Example 258. 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 483. Result

dist
1269.9148706779097
Rows: 1

Example 259. 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 484. Result

travelDistance
27842.0
Rows: 1

Example 260. 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 485. 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
point	A point in either a geographic or cartesian coordinate system.
lowerLeft	A point in the same CRS as 'point'.
upperRight	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 261. `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 486. Result

result
true
Rows: 1

Example 262. 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 487. Result

count(t)
1

Rows: 1

Example 263. 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 488. Result

result
true

Rows: 1

Example 264. 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 489. 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 ' <code>WGS-84</code> '.
<code>srid</code>	The optional number <code>4326</code> .

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 265. `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 490. Result

point
point({x: 56.7, y: 12.78, crs: 'wgs-84'})
Rows: 1

Example 266. `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 491. Result

point
point({x: 2.3, y: 4.5, crs: 'wgs-84'})
Rows: 1

Example 267. 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 492. Result

officePoint
point({x: 12.994341, y: 55.611784, crs: 'wgs-84'})
Rows: 1

Example 268. point()

Query

```
RETURN point(null) AS p
```

If `null` is provided as the argument, `null` is returned.

Table 493. Result

p
<null>
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
A single map consisting of the following:	

Name	Description
longitude/x	A numeric expression that represents the longitude/x value in decimal degrees.
latitude/y	A numeric expression that represents the latitude/y value in decimal degrees.
height/z	A numeric expression that represents the height/z value in meters.
crs	The optional string 'WGS-84-3D'.
srid	The optional number 4979.

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 269. `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 494. Result

point
point({x: 56.7, y: 12.78, z: 8.0, crs: 'wgs-84-3d'})
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:	
x	A numeric expression.
y	A numeric expression.
crs	The optional string 'cartesian'.
srid	The optional number 7203.

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 270. `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 495. Result

point
point({x: 2.3, y: 4.5, crs: 'cartesian'})

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 271. `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 496. Result

point
point({x: 2.3, y: 4.5, z: 2.0, crs: 'cartesian-3d'})
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.

Graph functions

graph.names()

Returns a list containing the names of all graphs on the current composite database. It is only supported on [composite databases](#).

Example 272. graph.names()

Setup

```
CREATE DATABASE dba;
CREATE DATABASE dbb;
CREATE DATABASE dbc;
CREATE COMPOSITE DATABASE composite;
CREATE ALIAS composite.first FOR DATABASE dba;
CREATE ALIAS composite.second FOR DATABASE dbb;
CREATE ALIAS composite.third FOR DATABASE dbc;
```

Query

```
RETURN graph.names() AS name
```

The names of all graphs on the current composite database are returned.

Table 497. Result

name
"composite.first"
"composite.second"
"composite.third"
Rows: 3

graph.propertiesByName()

Returns a map containing the properties associated with the given graph. The properties are set on the [alias](#) that adds the graph as a constituent of a composite database. It is only supported on [composite databases](#).

Example 273. graph.propertiesByName()

Setup

```
CREATE DATABASE dba;
CREATE DATABASE dbb;
CREATE DATABASE dbc;
CREATE COMPOSITE DATABASE composite;
CREATE ALIAS composite.first FOR DATABASE dba
    PROPERTIES {number: 1, tags: ['A', 'B']};
CREATE ALIAS composite.second FOR DATABASE dbb
    PROPERTIES {number: 0, tags: ['A']};
CREATE ALIAS composite.third FOR DATABASE dbc
    PROPERTIES {number: 2, tags: ['B', 'C']};
```

Query

```
UNWIND graph.names() AS name
RETURN name, graph.propertiesByName(name) AS props
```

Properties for all graphs on the current composite database are returned.

Table 498. Result

name	props
"composite.first"	{number: 1, tags: ["A", "B"]}
"composite.second"	{number: 0, tags: ["A"]}
"composite.third"	{number: 2, tags: ["B", "C"]}
Rows: 3	

graph.byName()

Resolves a constituent graph by name. It is only supported in the [USE clause](#), on composite databases.

Example 274. graph.byName()

Query

```
UNWIND graph.names() AS graphName
CALL {
    USE graph/byName(graphName)
    MATCH (n)
    RETURN n
}
RETURN n
```

Returns all nodes from all graphs on the current composite database.

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.	Using UDF	Extending Neo4j (UDF)
Aggregating	Consumes many rows and produces an aggregated result.	Using aggregating UDF	Extending Neo4j (Aggregating UDF)

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 275. 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.function.example.join(collect(n.name)) AS members
```

Table 499. Result

members

"John,Paul,George,Ringo"

Rows: 1

User-defined aggregation functions

Aggregating functions consume many rows and produces a single aggregated result.

Example 276. 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 500. 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:

- Range index.
- Lookup index.
- Text index.
- Point index.
- Full-text index.

See [Full-text search index](#) for more information about full-text indexes. Lookup indexes contain nodes with one or more labels or relationship types, without regard for any properties.

Cypher enables the creation of range 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, text and point indexes are a kind of single-property indexes, with the limitation that they only recognize properties with string and point values, respectively. 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.
 i	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.
 i	The CREATE ... INDEX ... command is optionally idempotent. This means that its default behavior is to throw an error if an attempt is made 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.
 i	The syntax descriptions use the style from access control.

Table 501. 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 OPTIONS clause.

Table 502. 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	<p>Create a range index on relationships, either on a single property or composite.</p> <p>Index provider can be specified using the OPTIONS clause.</p>
-------------	---

Table 503. 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 OPTIONS clause.</p>

Table 504. 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 OPTIONS clause.</p>

Table 505. 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 OPTIONS clause.</p>

Table 506. 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 OPTIONS clause.</p>
-------------	--

Table 507. 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 OPTIONS clause.</p>

Table 508. 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 OPTIONS clause.</p>

Table 509. 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. This means that its default behavior is to throw an error if an attempt is made to drop the same index twice. With IF EXISTS, no error is thrown and nothing happens should the index not exist.</p>

Table 510. List indexes

Syntax	<pre>SHOW [ALL 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>List indexes in the database, either all or filtered on index type.</p>

Note	When using the <code>RETURN</code> clause, the <code>YIELD</code> 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 range indexes, composite range 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`



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.



The `suffix search (ENDS WITH)` and `substring search (CONTAINS)` predicates can utilize the index as well. However, they are always planned as an 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 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 label lookup index
- Create a relationship type lookup index
- Create a token lookup index specifying the index provider
- 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 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 range index for nodes

A named range index on a single property for all nodes with a particular label can be created with:

```
CREATE INDEX index_name FOR (n:Label) ON (n.property)
```

Note that the index is not immediately available, but is created in the background.

Example 277. CREATE INDEX

Query

```
CREATE INDEX node_range_index_name FOR (n:Person) ON (n.surname)
```



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 INDEX index_name FOR ()-[r:TYPE]-() ON (r.property)
```

Note that the index is not immediately available, but is created in the background.

Example 278. CREATE INDEX

Query

```
CREATE INDEX rel_range_index_name FOR ()-[r:KNOWS]-() ON (r.since)
```



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 279. CREATE RANGE INDEX

Query

```
CREATE INDEX node_range_index_name IF NOT EXISTS  
FOR (n:Person) ON (n.surname)
```



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 280. CREATE INDEX

Query

```
CREATE INDEX range_index_with_provider
FOR ()-[r:TYPE]-()
ON (r.prop1)
OPTIONS {
    indexProvider: 'range-1.0'
}
```

Result

```
+-----+
| No data returned. |
+-----+
Indexes added: 1
```

There is no supported index configuration for range indexes.

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 INDEX index_name FOR (n:Label) ON (n.prop1, ..., n.propN)
```

Only nodes with the specified label and that contain all the properties in the index definition will be added to the index. Note that the composite index is not immediately available, but is created in the background.

Example 281. CREATE 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 INDEX composite_range_node_index_name FOR (n:Person) ON (n.age, n.country)
```



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 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 282. CREATE 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 INDEX composite_range_rel_index_name FOR ()-[r:PURCHASED]-() ON (r.date, r.amount)
```



The index name must be unique.

Result

```
+-----+  
| No data returned. |  
+-----+  
Indexes added: 1
```

Create a node label lookup index

A named node label 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 283. CREATE LOOKUP INDEX

Query

```
CREATE LOOKUP INDEX node_label_lookup_index FOR (n) ON EACH labels(n)
```



Note that a node label lookup index can only be created once and that the index name must be unique.

Result

```
+-----+  
| No data returned. |  
+-----+  
Indexes added: 1
```



Only one node label lookup index can exist at the time.

Create a relationship type lookup index

A named relationship type 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 284. CREATE LOOKUP INDEX

Query

```
CREATE LOOKUP INDEX rel_type_lookup_index FOR ()-[r]-() ON EACH type(r)
```



Note that a relationship type lookup index can only be created once and that the index name must be unique.

Result

```
+-----+  
| No data returned. |  
+-----+  
Indexes added: 1
```



Only one relationship type lookup index can exist at the time.

Create a token lookup index specifying the index provider

Token lookup indexes (node label and relationship type 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 285. CREATE LOOKUP INDEX

Query

```
CREATE LOOKUP INDEX node_label_lookup_index_2 FOR (n) ON EACH labels(n)
OPTIONS {indexProvider: 'token-lookup-1.0'}
```



Note that the above command will fail if any node label lookup index already exists.

Result

```
+-----+
| No data returned. |
+-----+
Indexes added: 1
```

There is no supported index configuration for token lookup indexes.

Create a node point index

A named point index on a single property for all nodes with a particular label can be created with:

```
CREATE POINT INDEX index_name FOR (n:Label) ON (n.property)
```

Note that the index is not immediately available, but is created in the background.

Example 286. CREATE POINT INDEX

Query

```
CREATE POINT INDEX node_index_name FOR (n:Person) ON (n.sublocation)
```



The index name must be unique.

Result

```
+-----+
| No data returned. |
+-----+
Indexes added: 1
```



Note that point indexes only recognize point values and do not support multiple properties.

Create a relationship point index

A named point index on a single property for all relationships with a particular relationship type can be created with:

```
CREATE POINT INDEX index_name FOR ()-[r:TYPE]-() ON (r.property)
```

Note that the index is not immediately available, but is created in the background.

Example 287. CREATE POINT INDEX

Query

```
CREATE POINT INDEX rel_index_name FOR ()-[r:STREET]-() ON (r.intersection)
```



The index name must be unique.

Result

```
+-----+  
| No data returned. |  
+-----+  
Indexes added: 1
```



Note that point indexes only recognize point values and do not support multiple properties.

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 288. CREATE POINT INDEX

Query

```
CREATE POINT INDEX node_index_name IF NOT EXISTS  
FOR (n:Person) ON (n.sublocation)
```



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 289. CREATE POINT INDEX

Query

```
CREATE POINT INDEX index_with_provider
FOR (n:Label) ON (n.prop1)
OPTIONS {
    indexProvider: 'point-1.0'
}
```

Result

```
+-----+
| No data returned. |
+-----+
Indexes added: 1
```

Specifying the index provider can be combined with specifying index configuration.

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 290. 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]
    }
}
```

Result

```
+-----+
| No data returned. |
+-----+
Indexes added: 1
```

Specifying the index configuration can be combined with specifying index provider.

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 291. 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]
    }
}
```

Result

```
+-----+
| No data returned. |
+-----+
Indexes added: 1
```

Index provider and configuration can also be specified separately.

Create a node text index

A named text index on a single property for all nodes with a particular label can be created with:

```
CREATE TEXT INDEX index_name FOR (n:Label) ON (n.property)
```



The index is not immediately available, but is created in the background.

Example 292. CREATE TEXT INDEX

Query

```
CREATE TEXT INDEX node_index_name FOR (n:Person) ON (n.nickname)
```



The index name must be unique.

Result

```
+-----+
| No data returned. |
+-----+
Indexes added: 1
```



Text indexes only recognize string values and do not support multiple properties.

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 293. CREATE TEXT INDEX

Query

```
CREATE TEXT INDEX rel_index_name FOR ()-[r:KNOWS]-() ON (r.interest)
```



The index name must be unique.

Result

```
+-----+  
| No data returned. |  
+-----+  
Indexes added: 1
```



Note that text indexes only recognize string values and do not support multiple properties.

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 294. 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. The valid values for the index provider are `text-2.0` and `text-1.0` (deprecated). The default provider is `text-2.0`.

Example 295. CREATE TEXT INDEX

Query

```
CREATE TEXT INDEX index_with_provider FOR ()-[r:TYPE]-() ON (r.prop1)
OPTIONS {indexProvider: 'text-2.0'}
```

Result

```
+-----+
| No data returned. |
+-----+
Indexes added: 1
```

There is no supported index configuration for text indexes.

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 296. CREATE RANGE 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 the given name already exists.

Example 297. CREATE RANGE 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 the given 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 298. CREATE RANGE 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 the given name already exists.

Example 299. CREATE RANGE 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 the given 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:

Table 511. List indexes output

Column	Description
<code>id</code>	The id of the index. Default output
<code>name</code>	Name of the index (explicitly set by the user or automatically assigned). Default output
<code>state</code>	Current state of the index. Default output
<code>populationPercent</code>	% of index population. Default output
<code>type</code>	The IndexType of this index (<code>FULLTEXT</code> , <code>LOOKUP</code> , <code>POINT</code> , <code>RANGE</code> , or <code>TEXT</code>). Default output
<code>owningConstraint</code>	The name of the constraint the index is associated with or <code>null</code> , in case it is not associated with any constraint. Default output
<code>entityType</code>	Type of entities this index represents (nodes or relationship). Default output
<code>labelsOrTypes</code>	The labels or relationship types of this index. Default output
<code>properties</code>	The properties of this index. Default output
<code>indexProvider</code>	The index provider for this index. Default output
<code>options</code>	The options passed to <code>CREATE</code> command.
<code>failureMessage</code>	The failure description of a failed index.
<code>createStatement</code>	Statement used to create the index.



The command `SHOW INDEXES` returns only the default output. For a full output use the optional `YIELD` command. Full output: `SHOW INDEXES YIELD *`.

Listing indexes also allows for `WHERE` and `YIELD` clauses to filter the returned rows and columns.

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 300. `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 | type    | entityType | 
| labelsOrTypes | properties      | indexProvider |          | owningConstraint |
+-----+-----+-----+-----+-----+-----+
| 10 | "constraint_1bc95fcb" | "ONLINE" | 100.0          | "RANGE" | "NODE"     | ["Person"]
| ["name"] | "range-1.0" |          | "constraint_1bc95fcb" |
| 4 | "index_58a1c03e" | "ONLINE" | 100.0          | "RANGE" | "NODE"     | ["Person"]
| ["location"] | "point-1.0" |          | <null>
| 3 | "index_664b28a2" | "ONLINE" | 100.0          | "RANGE" | "NODE"     | ["Person"]
| ["middlename"] | "range-1.0" |          | <null>
| 7 | "index_763f72db" | "ONLINE" | 100.0          | "TEXT"  | "NODE"     | ["Person"]
| ["middlename"] | "text-1.0" |          | <null>
| 5 | "index_8a688dca" | "ONLINE" | 100.0          | "RANGE" | "NODE"     | ["Person"]
| ["highScore"] | "range-1.0" |          | <null>
| 6 | "index_b87724c3" | "ONLINE" | 100.0          | "RANGE" | "NODE"     | ["Person"]
| ["firstname"] | "range-1.0" |          | <null>
| 9 | "index_c3493fe4" | "ONLINE" | 100.0          | "POINT" | "NODE"     | ["Person"]
| ["location"] | "point-1.0" |          | <null>
| 1 | "index_d7c12ba3" | "ONLINE" | 100.0          | "LOOKUP" | "NODE"     | <null>
| <null> | "token-lookup-1.0" | <null>
| 2 | "index_deeafdb2" | "ONLINE" | 100.0          | "LOOKUP" | "RELATIONSHIP" | <null>
| <null> | "token-lookup-1.0" | <null>
| 8 | "index_eadb868e" | "ONLINE" | 100.0          | "TEXT"  | "NODE"     | ["Person"]
| ["surname"] | "text-1.0" |          | <null>
+-----+-----+-----+-----+-----+-----+
                                         |
                                         7 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 range indexes, use `SHOW RANGE 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 301. SHOW RANGE INDEXES

Query

```
SHOW RANGE INDEXES WHERE owningConstraint IS NULL
```

This will only return the default output columns.

To get all columns, use:

```
SHOW INDEXES YIELD * WHERE ...
```

Result

```
+-----+
| id | name           | state      | populationPercent | type       | entityType     | labelsOrTypes |
+-----+
| 3  | "index_664b28a2" | "ONLINE"   | 100.0        | "RANGE"    | "NODE"         | ["Person"]    |
| 6  | "index_6e62c571" | "ONLINE"   | 100.0        | "RANGE"    | "RELATIONSHIP" | ["KNOWS"]    |
| 4  | "index_8a688dca" | "ONLINE"   | 100.0        | "RANGE"    | "NODE"         | ["Person"]    |
| 5  | "index_b87724c3" | "ONLINE"   | 100.0        | "RANGE"    | "NODE"         | ["Person"]    |
+-----+
4 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 302. 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 303. DROP INDEX

Query

```
DROP INDEX missing_index_name IF EXISTS
```

Result

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

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 range and text indexes described in previous sections can only perform limited matching on strings; exact, prefix, substring, or suffix matches. 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.
- include only property values of types String or String Array.
- 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 [other 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 full-text index for the given labels and properties.
Create full-text relationship index.	<code>CREATE FULLTEXT INDEX ...</code>	Create a relationship full-text index for the given relationship types and properties.
List available analyzers.	<code>db.index.fulltext.listAvailableAnalyzers</code>	List the available analyzers that the full-text indexes can be configured with.
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 full-text indexes.	<code>SHOW FULLTEXT INDEXES</code>	Lists all full-text indexes, see the SHOW INDEXES command 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.



The syntax descriptions use the style from access control.

Table 512. Syntax for creating full-text indexes

Command	Description
<pre>CREATE FULLTEXT INDEX [index_name] [IF NOT EXISTS] FOR (n:LabelName[" " ...]) ON EACH "[" n.propertyName[, ...] "]" [OPTIONS "{" option: value[, ...] "}"]</pre>	Create a full-text index on nodes.
<pre>CREATE FULLTEXT INDEX [index_name] [IF NOT EXISTS] FOR ()-["r:TYPE_NAME[" " ..."]"]-() ON EACH "[" r.propertyName[, ...] "]" [OPTIONS "{" option: value[, ...] "}"]</pre>	Create a full-text index on relationships.

It is considered best practice to give the index a name when it is created. This name is needed for both dropping and querying the index. If the index is not explicitly named, it will get an auto-generated name.



The index name must be unique among all indexes and constraints.

Index provider and configuration settings can be specified using the `OPTIONS` clause. Supported settings are `fulltext.analyzer`, for specifying what analyzer to use when indexing and querying. Use the `db.index.fulltext.listAvailableAnalyzers` procedure to see what options are available. To make this index eventually consistent, `fulltext.eventually_consistent` should be set to `true`. This will ensure that updates from committing transactions are applied in a background thread.

The command is optionally idempotent. This means that its default behavior is to throw an error if an attempt is made to create the same index twice. With `IF NOT EXISTS`, no error is thrown and nothing happens should an index with the same name, schema or both already exist. It may still throw an error should a constraint with the same name exist.

Example 304. CREATE FULLTEXT INDEX

For instance, if we have a movie with a title.

Query

```
CREATE (m:Movie {title: "The Matrix"}) RETURN m.title
```

Table 513. 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 514. 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 takes an optional clause, called `options`. This has 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 305. 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 515. 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 306. 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 516. 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 307. 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 517. 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 308. 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 518. 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 309. Query full-text

Query

```
CALL db.index.fulltext.queryNodes("titlesAndDescriptions", 'description:surreal adventure') YIELD
node, score
RETURN node.title, node.description, score
```

Table 519. 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](#).

Handling of Text Array properties

If the indexed property contains a text array, each element of this array is analyzed independently and all produced terms are associated with the same property name. This means that when querying such an indexed node or relationship, there is a match if any of the array elements match the query. For scoring purposes, the full-text index treats it as a single-property value, and the score will represent how close the query is to matching the entire array.

Example 310. Text Array properties

For example, both of the following queries match the same node while referring different elements:

Query

```
CALL db.index.fulltext.queryNodes("reviews", 'best') YIELD node, score
RETURN
node.title AS title,
node.reviews AS reviews,
score
```

Result

Rows: 1

title	reviews	score
'The Matrix'	['The best movie ever.', 'The movie is nonsense.']}	0.13076457381248474

Query

```
CALL db.index.fulltext.queryNodes("reviews", 'nonsense') YIELD node, score
RETURN
node.title AS title,
node.reviews AS reviews,
score
```

Result

Rows: 1

title	reviews	score
'The Matrix'	['The best movie ever.', 'The movie is nonsense.']}	0.13076457381248474

Drop full-text indexes

A full-text node index is dropped by using the [same command as for other indexes](#), `DROP INDEX`.

Example 311. DROP INDEX

In the following example, we will drop the `taggedByRelationshipIndex` that we created previously:

Query

```
DROP INDEX taggedByRelationshipIndex
```

Table 520. 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 node property constraints, or node property uniqueness constraints, ensure that property values are unique for all nodes with a specific label. For property uniqueness constraints on multiple properties, the combination of the property values is unique. Node property uniqueness 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

Relationship 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 property uniqueness 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 property uniqueness 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 property uniqueness 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 uniqueness 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



The syntax descriptions use [the style](#) from access control.

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. This means its default behavior is to throw an error if an attempt is made 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 conflicting constraint type.

For constraints that are backed by an index, the index provider for the backing index can be specified using the `OPTIONS` clause. Only one valid value exists for the index provider, `range-1.0`, which is the default value. There is no supported index configuration for range indexes.



Creating a constraint requires the `CREATE CONSTRAINT` privilege.

Create a node property uniqueness constraint

This command creates a property uniqueness constraint on nodes with the specified label and properties.

```
CREATE CONSTRAINT [constraint_name] [IF NOT EXISTS]
FOR (n:LabelName)
REQUIRE n.propertyName IS [NODE] UNIQUE
[OPTIONS "{" option: value[, ...] "}" ]
```

```
CREATE CONSTRAINT [constraint_name] [IF NOT EXISTS]
FOR (n:LabelName)
REQUIRE (n.propertyName_1, ..., n.propertyName_n) IS [NODE] UNIQUE
[OPTIONS "{" option: value[, ...] "}" ]
```

Index provider 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 can be specified using the `OPTIONS` clause.

Syntax for dropping constraints

Dropping a constraint is done by specifying the name of the constraint.

```
DROP CONSTRAINT constraint_name [IF EXISTS]
```

This drop command is optionally idempotent. This means its default behavior is to throw an error if an attempt is made 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.

Syntax for listing constraints

List constraints in the database, either all or filtered on constraint type.



Listing constraints requires the `SHOW CONSTRAINTS` privilege.

The simple version of the command allows for a `WHERE` clause and will give back the default set of output columns:

```
SHOW [
    ALL
    |UNIQUENESS]
    |NODE [PROPERTY] EXISTENCE
    |RELATIONSHIP [PROPERTY] EXISTENCE
    |[PROPERTY] EXISTENCE
    |NODE KEY
] CONSTRAINT[S]
[WHERE expression]
```

To get the full set of output columns, a yield clause is needed:

```

SHOW [  

    ALL  

    |UNIQUENESS]  

|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 type filtering keywords filters the returned constraints on constraint type:

Table 521. Type filters

Filter	Description
ALL	Returns all constraints, no filtering on constraint type. This is the default if none is given.
UNIQUENESS	Returns all property uniqueness constraints.
NODE [PROPERTY] EXIST[ENCE]	Returns the node property existence constraints.
REL[ATIONSHIP] [PROPERTY] EXIST[ENCE]	Returns the relationship property existence constraints.
[PROPERTY] EXIST[ENCE]	Returns all property existence constraints, for both nodes and relationships.
NODE KEY	Returns the node key constraints.

The returned columns from the show command is:

Table 522. Listing constraints output

Column	Description
id	The id of the constraint. Default output
name	Name of the constraint (explicitly set by the user or automatically assigned). Default output
type	The ConstraintType of this constraint (UNIQUENESS , NODE_PROPERTY_EXISTENCE , RELATIONSHIP_PROPERTY_EXISTENCE , or NODE_KEY). Default output
entityType	Type of entities this constraint represents (nodes or relationship). Default output

Column	Description
<code>labelsOrTypes</code>	The labels or relationship types of this constraint. Default output
<code>properties</code>	The properties of this constraint. Default output
<code>ownedIndex</code>	The name of the index associated with the constraint or <code>null</code> , in case no index is associated with it. Default output
<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.

Examples

Examples of how to manage constraints used for ensuring data integrity.

Node property uniqueness constraints

A node property uniqueness constraint ensures that all nodes with a particular label have a set of defined properties whose combined value is unique when existing.

- [Create a node property uniqueness constraint](#)
- [Handling existing constraints when creating a constraint](#)
- [Specifying an index provider when creating a constraint](#)
- [Creating an already existing constraint will fail](#)
- [Creating a constraint on the same schema as an existing index will fail](#)
- [Creating a node that complies with an existing constraint](#)
- [Creating a node that violates an existing constraint will fail](#)
- [Creating a constraint when there exist conflicting nodes will fail](#)

Create a node property uniqueness constraint

When creating a property uniqueness constraint, a name can be provided.

Example 312. CREATE CONSTRAINT

Query

```
CREATE CONSTRAINT constraint_name
FOR (book:Book) REQUIRE book.isbn IS UNIQUE
```

Result

```
+-----+
| No data returned. |
+-----+
Unique constraints added: 1
```

Handling existing constraints when creating a constraint

Creating an already existing constraint will fail. To avoid such an error, `IF NOT EXISTS` can be added to the `CREATE` command. This will ensure that no error is thrown and no constraint is created if any other constraint with the given name or another node property uniqueness constraint on the same schema already exists.

Example 313. CREATE CONSTRAINT

Query

```
CREATE CONSTRAINT constraint_name IF NOT EXISTS
FOR (book:Book) REQUIRE book.isbn IS UNIQUE
```

Assuming no constraint with the given name or other node property uniqueness constraint on the same schema exists:

Result

```
+-----+
| No data returned. |
+-----+
Unique constraints added: 1
```

Specifying an index provider when creating a constraint

To create a property uniqueness constraint with a specific index provider for the backing index, the `OPTIONS` clause is used.

The index type of the backing index is set with the `indexProvider` option.

The only valid value for the index provider is:

- range-1.0 **Default**

Example 314. CREATE CONSTRAINT

Query

```
CREATE CONSTRAINT constraint_with_options
FOR (n:Label) REQUIRE (n.prop1, n.prop2) IS UNIQUE
OPTIONS {
    indexProvider: 'range-1.0',
}
```

Result

```
+-----+
| No data returned. |
+-----+
Unique constraints added: 1
```

There is no valid index configuration values for the constraint-backing range indexes.

Creating an already existing constraint will fail

Example 315. CREATE CONSTRAINT

Create a property uniqueness 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 )
```

Creating a constraint on the same schema as an existing index will fail

Example 316. CREATE CONSTRAINT

Create a property uniqueness 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.
```

Creating a node that complies with an existing constraint

Example 317. CREATE NODE

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
```

Creating a node that violates an existing constraint will fail

Example 318. CREATE NODE

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'
```

Creating a constraint when there exist conflicting nodes will fail

Example 319. CREATE CONSTRAINT

Create a property uniqueness 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. Either 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

A node property existence constraint ensures that all nodes with a certain label have a certain property.

- [Create a node property existence constraint](#)
- [Handling existing constraints when creating a constraint](#)
- [Creating an already existing constraint will fail](#)
- [Creating a node that complies with an existing constraint](#)
- [Creating a node that violates an existing constraint will fail](#)
- [Removing an existence constrained node property will fail](#)
- [Creating a constraint when there exist conflicting nodes will fail](#)

Create a node property existence constraint

When creating a node property existence constraint, a name can be provided.

Example 320. 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
```

Handling existing constraints when creating a constraint

Creating an already existing constraint will fail. To avoid such an error, `IF NOT EXISTS` can be added to the `CREATE` command. This will ensure that no error is thrown and no constraint is created if any other constraint with the given name or another node property existence constraint on the same schema already existed.

Example 321. 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. |  
+-----+
```

Creating an already existing constraint will fail

Example 322. 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}) )
```

Creating a node that complies with an existing constraint

Example 323. CREATE NODE

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
```

Creating a node that violates an existing constraint will fail

Example 324. CREATE NODE

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 will fail

Example 325. REMOVE PROPERTY

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

Query

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

In this case the property is not removed.

Error message

```
Node(0) with label `Book` must have the property `isbn`
```

Creating a constraint when there exist conflicting nodes will fail

Example 326. 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. 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

A relationship property existence constraint ensures that all relationships with a certain type have a certain property.

- [Create a relationship property existence constraint](#)
- [Handling existing constraints when creating a constraint](#)
- [Creating an already existing constraint will fail](#)
- [Creating a relationship that complies with an existing constraint](#)
- [Creating a relationship that violates an existing constraint will fail](#)
- [Removing an existence constrained relationship property will fail](#)
- [Creating a constraint when there exist conflicting relationships will fail](#)

Create a relationship property existence constraint

When creating a relationship property existence constraint, a name can be provided.

Example 327. 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
```

Handling existing constraints when creating a constraint

Creating an already existing constraint will fail. To avoid such an error, `IF NOT EXISTS` can be added to the `CREATE` command. This will ensure that no error is thrown and no constraint is created if any other constraint with the given name or another relationship property existence constraint on the same schema already existed.

Example 328. 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. |  
+-----+
```

Creating an already existing constraint will fail

Example 329. CREATE CONSTRAINT

Create a named relationship property existence constraint on the property `week` on relationships with the `LIKED` type, when a constraint with the given 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 the given name.

Error message

```
There already exists a constraint called 'relPropExist'.
```

Creating a relationship that complies with an existing constraint

Example 330. CREATE RELATIONSHIP

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
```

Creating a relationship that violates an existing constraint will fail

Example 331. CREATE RELATIONSHIP

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 will fail

Example 332. REMOVE PROPERTY

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

Query

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

In this case the property is not removed.

Error message

```
Relationship(0) with type `LIKED` must have the property `day`
```

Creating a constraint when there exist conflicting relationships will fail

Example 333. 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. 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

A 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.

- [Create a node key constraint](#)
- [Handling existing constraints when creating a constraint](#)
- [Specifying an index provider when creating a constraint](#)
- [Node key and property uniqueness constraints are not allowed on the same schema](#)
- [Creating a constraint on same name as an existing index will fail](#)
- [Creating a node that complies with an existing constraint](#)
- [Creating a node that violates an existing constraint will fail](#)

- Removing a NODE KEY-constrained property will fail
- Creating a constraint when there exist conflicting node will fail

Create a node key constraint

When creating a node key constraint, a name can be provided.

Example 334. CREATE CONSTRAINT

Query

```
CREATE CONSTRAINT constraint_name
FOR (:Person) REQUIRE (n.firstname, n.surname) IS NODE KEY
```

Result

```
+-----+
| No data returned. |
+-----+
Node key constraints added: 1
```

Handling existing constraints when creating a constraint

Creating an already existing constraint will fail. To avoid such an error, `IF NOT EXISTS` can be added to the `CREATE` command. This will ensure that no error is thrown and no constraint is created if any other constraint with the given name or another node key constraint on the same schema already exists.

Example 335. CREATE CONSTRAINT

Query

```
CREATE CONSTRAINT constraint_name IF NOT EXISTS
FOR (: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. |
+-----+
```

Specifying an index provider when creating a constraint

To create a node key constraint with a specific index provider for the backing index, the `OPTIONS` clause is used.

The index type of the backing index is set with the `indexProvider` option.

The only valid value for the index provider is:

- range-1.0 Default

Example 336. CREATE CONSTRAINT

Query

```
CREATE CONSTRAINT constraint_with_provider
FOR (n:Label) REQUIRE (n.prop1) IS NODE KEY
OPTIONS {
    indexProvider: 'range-1.0'
}
```

Result

```
+-----+
| No data returned. |
+-----+
Node key constraints added: 1
```

There is no valid index configuration values for the constraint-backing range indexes.

Node key and property uniqueness constraints are not allowed on the same schema

Example 337. CREATE CONSTRAINT

Create a node key constraint on the properties `firstname` and `age` on nodes with the `Person` label, when a property uniqueness 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 )
```

Creating a constraint on same name as an existing index will fail

Example 338. CREATE CONSTRAINT

Create a named node key constraint on the property `title` on nodes with the `Book` label, when an index already exists with the given 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 the given name.

Error message

```
There already exists an index called 'bookTitle'.
```

Creating a node that complies with an existing constraint

Example 339. CREATE NODE

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
```

Creating a node that violates an existing constraint will fail

Example 340. CREATE NODE

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 will fail

Example 341. REMOVE PROPERTY

Trying to remove the `surname` property from an existing node `Person`, given a `NODE KEY` constraint on `:Person(firstname, surname)`.

Query

```
MATCH (p:Person {firstname: 'John', surname: 'Wood'}) REMOVE p.surname
```

In this case the property is not removed.

Error message

```
Node(0) with label 'Person' must have the properties ('firstname', 'surname')
```

Creating a constraint when there exist conflicting node will fail

Example 342. 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 (: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. Either use [Indexes for search performance](#) instead, or remove the offending nodes and then re-apply the constraint.

Error message

```
Unable to create Constraint( type='NODE KEY', 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 uniqueness, 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 343. 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 uniqueness, property existence, and node constraints.

Example 344. 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 345. SHOW CONSTRAINTS

Query

```
SHOW CONSTRAINTS
```

```
+-----+  
| id | name          | type        | entityType | labelsOrTypes | properties | ownedIndex |  
+-----+  
| 4  | "isbnConstraint" | "UNIQUENESS" | "NODE"     | ["Book"]    | ["isbn"]   | "isbnConstraint" |  
+-----+  
2 rows
```

Listing constraints with filtering

One way of filtering the output from `SHOW CONSTRAINTS` by constraint type is the use of type keywords, listed in the [syntax for listing constraints type filter table](#). For example, to show only property uniqueness constraints, use `SHOW UNIQUENESS 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 346. 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

```
....
```

id	name	type	entityType	labelsOrTypes
properties	ownedIndex			
7	"constraint_f076a74d"	"RELATIONSHIP_PROPERTY_EXISTENCE"	"RELATIONSHIP"	["KNOWS"]
["since"]	<null>			

1 row

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:



The syntax descriptions use the style from access control.

Table 523. Database management command syntax

Command	Syntax
SHOW DATABASE	<pre>SHOW { DATABASE[S] name DATABASE[S] DEFAULT DATABASE HOME DATABASE } [WHERE expression]</pre> <pre>SHOW { DATABASE[S] name DATABASE[S] 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>
CREATE DATABASE	<pre>CREATE DATABASE name [IF NOT EXISTS] [TOPOLOGY n PRIMAR{Y IES} [m SECONDAR{Y IES}]] [OPTIONS "{" option: value[, ...] "}] [WAIT [n [SECOND[S]]]] NOWAIT]</pre> <pre>CREATE OR REPLACE DATABASE name [TOPOLOGY n PRIMAR{Y IES} [m SECONDAR{Y IES}]] [OPTIONS "{" option: value[, ...] "}] [WAIT [n [SECOND[S]]]] NOWAIT]</pre>
CREATE COMPOSITE DATABASE	<pre>CREATE COMPOSITE DATABASE name [IF NOT EXISTS] [OPTIONS {" "}] [WAIT [n [SECOND[S]]]] NOWAIT]</pre> <pre>CREATE OR REPLACE COMPOSITE DATABASE name [OPTIONS {" "}] [WAIT [n [SECOND[S]]]] NOWAIT]</pre>

Command	Syntax
ALTER DATABASE	<pre>ALTER DATABASE name [IF EXISTS] { SET ACCESS {READ ONLY READ WRITE} SET TOPOLOGY n PRIMAR{Y IES} [m SECONDAR{Y IES}] }</pre>
STOP DATABASE	<pre>STOP DATABASE name [WAIT [n [SECOND[S]]]] NOWAIT]</pre>
START DATABASE	<pre>START DATABASE name [WAIT [n [SECOND[S]]]] NOWAIT]</pre>
DROP DATABASE	<pre>DROP [COMPOSITE] DATABASE name [IF EXISTS] [{DUMP DESTROY} [DATA]] [WAIT [n [SECOND[S]]]] NOWAIT]</pre>

List 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 524. Listing databases output

Column	Description
name	The name of the database. Default output
type	The type of the database: <code>system</code> , <code>standard</code> , or <code>composite</code> . Default output
aliases	The names of any aliases the database may have. Default output

Column	Description	
access	The database access mode, either <code>read-write</code> or <code>read-only</code> . Default output	
		A database may be described as <code>read-only</code> when using <code>ALTER DATABASE ... SET ACCESS READ ONLY</code> .
databaseID	The database unique ID.	
serverID	The server instance ID.	
address	Instance address in a clustered DBMS. The default for a standalone database is <code>neo4j://localhost:7687</code> . Default output	
role	The current role of the database (<code>primary</code> , <code>secondary</code> , <code>unknown</code>). Default output	
writer	<code>true</code> for the database node that accepts writes (this node is the leader for this database in a cluster or this is a standalone instance). Default output	
requestedStatus	The expected status of the database. Default output	
currentStatus	The actual status of the database. Default output	
error	An error message explaining why the database is not in the correct state. Default output	
default	Show if this is the default database for the DBMS. Default output	
		Not returned by <code>SHOW HOME DATABASE</code> or <code>SHOW DEFAULT DATABASE</code> .
home	Shown if this is the home database for the current user. Default output	
		Not returned by <code>SHOW HOME DATABASE</code> or <code>SHOW DEFAULT DATABASE</code> .

Column	Description
currentPrimariesCount	Number of primaries for this database reported as running currently. It is the same as the number of rows where <code>role=primary</code> and <code>name=this database</code> .
currentSecondariesCount	Number of secondaries for this database reported as running currently. It is the same as the number of rows where <code>role=secondary</code> and <code>name=this database</code> .
requestedPrimariesCount	The requested number of primaries for this database. May be lower than current if the DBMS is currently reducing the number of copies of the database, or higher if it is currently increasing the number of copies.
requestedSecondariesCount	The requested number of secondaries for this database. May be lower than current if the DBMS is currently reducing the number of copies of the database, or higher if it is currently increasing the number of copies.
creationTime	The date and time at which the database was created.
lastStartTime	The date and time at which the database was last started.
lastStopTime	The date and time at which the database was last stopped.
store	<p>Information about the storage engine and the store format.</p> <p>The value is a string formatted as:</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> {storage engine}-{store format}-{major version}.{minor version} </div>
lastCommittedTxn	The ID of the last transaction received.
replicationLag	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> .
constituents	<p>The names of any constituents the database may have.</p> <div style="border: 1px solid #0072bc; color: white; padding: 2px; margin-top: 5px;"> Default output </div>

Example 347. SHOW DATABASES

A summary of all available databases can be displayed using the command `SHOW DATABASES`.

Query

```
SHOW DATABASES
```

Table 525. 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, some privileges enable users to see additional databases regardless of their `ACCESS` privileges:

- Users with `CREATE/DROP/ALTER DATABASE` or `SET DATABASE ACCESS` privileges can see all standard databases.
- Users with `CREATE/DROP COMPOSITE DATABASE` or `COMPOSITE DATABASE MANAGEMENT` privileges can see all composite databases.
- Users with `DATABASE MANAGEMENT` privilege can see all databases.

If a user has not been granted `ACCESS` privilege to any databases nor any of the above special cases, the command can still be executed but will only return the `system` database, which is always visible.

 Databases hosted on servers that are offline are also returned by the `SHOW DATABASES` command. For such databases, the `address` column displays `NULL`, the `currentStatus` column displays `unknown`, and the `statusMessage` displays `Server is unavailable`.

Example 348. 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 526. Result

name	aliases	access	databaseID	serverID	address	...
"movies"	["films", "moticon pictures"]	"read-write"	"367221F9021C00CEBFCA25C8E2101F1DCF45C7DB9BF7D7A0949B87745E760EDD"	"adc0a7bc-d9a6-4cc8-b394-91635fbb1137"	"localhost:7687"	...

Rows: 1

Example 349. 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 527. Result

count
3

Rows: 1

Example 350. SHOW DEFAULT DATABASE

The default database can be seen using the command `SHOW DEFAULT DATABASE`.

Query

```
SHOW DEFAULT DATABASE
```

Table 528. Result

name	aliases	access	address	role	requestedStatus	currentStatus	error
"neo4j"	[]	"read-write"	"localhost:7687"	"standalone"	"online"	"online"	" "

Rows: 1

Example 351. SHOW HOME DATABASE

The home database for the current user can be seen using the command `SHOW HOME DATABASE`.

Query

```
SHOW HOME DATABASE
```

Table 529. Result

name	aliases	access	address	role	requestedStatus	currentStatus	error
"neo4j"	[]	"read-write"	"localhost:7687"	"standalone"	"online"	"online"	" "

Rows: 1

Example 352. 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 530. 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`.

For composite databases the `constituents` column is particularly interesting as it lists the aliases that make up the composite database:

Query

```
SHOW DATABASE library YIELD name, constituents
```

Table 531. Result

name	constituents
"library"	["library.sci-fi", "library.romance"]

Rows: 1

Creating databases

Enterprise edition

Databases can be created using `CREATE DATABASE`.

Example 353. `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``). Using database names with dots without enclosing them in backticks is deprecated.
- Names cannot end with dots or dashes.
- Names that begin with an underscore or with the prefix `system` are reserved for internal use.



Example 354. 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 532. 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

Cluster topology Enterprise edition

In a cluster environment, it may be desirable to control the number of servers used to host a database. The number of primary and secondary servers can be specified using the following command.

Query

```
CREATE DATABASE `topology-example` TOPOLOGY 1 PRIMARY 0 SECONDARIES
```

For more details on primary and secondary server roles, see [Cluster overview](#).



TOPOLOGY is only available for standard databases and not composite databases.
Composite databases are always available on all servers.

Creating composite databases Enterprise edition

Composite databases do not contain data, but they reference to other databases that can be queried together through their constituent aliases. For more information about composite databases, see [Operations Manual → Composite database introduction](#).

Composite databases can be created using `CREATE COMPOSITE DATABASE`.

Query

```
CREATE COMPOSITE DATABASE inventory
```

0 rows, System updates: 1



Composite database names are subject to the same rules as standard databases. One difference is however that the deprecated syntax using dots without enclosing the name in backticks is not available. Both dots and dashes needs to be enclosed within backticks when using composite databases.

When a composite database has been created, it will show up in the listing provided by the command `SHOW DATABASES`.

Query

```
SHOW DATABASES YIELD name, type, access, role, writer, constituents
```

Table 533. Result

name	type	access	role	writer	constituents
"customers"	"standard"	"read-write"	"primary"	true	[]
"inventory"	"composite"	"read-only"	<null>	false	[]
"library"	"composite"	"read-only"	<null>	false	["library.sci-fi", "library.romance"]
"movies"	"standard"	"read-write"	"primary"	true	[]
"neo4j"	"standard"	"read-write"	"primary"	true	[]
"sci-fi-books"	"standard"	"read-write"	"primary"	true	[]
"system"	"system"	"read-write"	"primary"	true	[]
"topology-example"	"standard"	"read-write"	"primary"	true	[]

Rows: 8

In order to create database aliases in the composite database, give the composite database as namespace for the alias. For information about creating aliases in composite databases, see [here](#).

Handling Existing Databases Enterprise edition

These commands are 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.

These behavior flags apply to both standard and composite databases (e.g. a composite database may replace a standard one or another composite.)

Example 355. CREATE DATABASE

Query

```
CREATE COMPOSITE DATABASE customers IF NOT EXISTS
```

Example 356. 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 these commands cannot be used together.

Options Enterprise edition

The `CREATE DATABASE` command can have a map of options, e.g. `OPTIONS {key: 'value'}`.



There are no available `OPTIONS` values for composite databases.

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 <code>id</code> column of the <code>dbms.cluster.overview()</code> procedure. Can only be used in clusters.
<code>seedURI</code>	URI to a backup or a dump from an existing database.	Defines an identical seed from an external source which will be used to seed all servers.
<code>seedConfig</code>	comma separated list of configuration values.	Defines additional configuration specified by comma separated <code>name=value</code> pairs that might be required by certain seed providers.

Key	Value	Description
seedCredentials	credentials	Defines credentials that needs to be passed into certain seed providers.



The `existingData`, `existingDataSeedInstance`, `seedURI`, `seedConfig` and `seedCredentials` options cannot be combined with the OR REPLACE part of this command. For details about the use of these seeding options, see [Operations Manual → Seed a cluster](#).

Altering databases Enterprise edition

Standard databases can be modified using the command `ALTER DATABASE`.

Access

By default, a database has read-write access mode on creation. The database can be limited to read-only mode on creation using the configuration parameters `dbms.databases.default_to_read_only`, `dbms.databases.read_only`, and `dbms.database.writable`. For details, see [Configuration parameters](#).

A database that was created with read-write access mode can be changed to read-only. 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.

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.



Modifying access mode is only available to standard databases and not composite databases.

Example 357. ALTER DATABASE

Query

```
ALTER DATABASE customers SET ACCESS READ ONLY
```

Result

```
System updates: 1
Rows: 0
```

Example 358. 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 534. Result

name	access
"customers"	"read-only"
"movies"	"read-write"
"neo4j"	"read-write"
"system"	"read-write"

Rows: 4

Example 359. ALTER DATABASE

`ALTER DATABASE` commands are 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
```

Topology

In a cluster environment, it may be desirable to change the number of servers used to host a database. The number of primary and secondary servers can be specified using the following command:

Example 360. ALTER DATABASE

Query

```
ALTER DATABASE `topology-example`  
SET TOPOLOGY 3 PRIMARY 0 SECONDARIES
```



It is not possible to automatically transition to or from a topology with a single primary host. See the [Operations Manual → Alter topology](#) for more information.

Example 361. SHOW DATABASE

Query

```
SHOW DATABASES yield name, currentPrimariesCount, currentSecondariesCount, requestedPrimariesCount,  
requestedSecondariesCount
```

For more details on primary and secondary server roles, see [Operations Manual → Clustering overview](#).



Modifying database topology is only available to standard databases and not composite databases.

`ALTER DATABASE` commands are 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 TOPOLOGY 1 PRIMARY 0 SECONDARY
```

0 rows

Stopping databases Enterprise edition

Databases can be stopped using the command `STOP DATABASE`.

Example 362. STOP DATABASE

Query

```
STOP DATABASE customers
```

Result

```
System updates: 1  
Rows: 0
```



Both standard databases and composite databases can be stopped using this command.

Example 363. SHOW DATABASE

The status of the stopped database can be seen using the command `SHOW DATABASE name`.

Query

```
SHOW DATABASE customers
```

Table 535. 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 364. START DATABASE

Query

```
START DATABASE customers
```

Result

```
System updates: 1  
Rows: 0
```



Both standard databases and composite databases can be stopped using this command.

Example 365. SHOW DATABASE

The status of the started database can be seen using the command `SHOW DATABASE name`.

Query

```
SHOW DATABASE customers
```

Table 536. 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

Standard and composite databases can be deleted by using the command `DROP DATABASE`.

Example 366. DROP DATABASE

Query

```
DROP DATABASE customers
```

Result

```
System updates: 1  
Rows: 0
```

It is also possible to ensure that only composite databases are dropped. A `DROP COMPOSITE` request would then fail if the targeted database is a standard database.

Example 367. 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 537. 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 368. 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 369. 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
```

It is also possible to ensure that only composite databases are dropped. A `DROP COMPOSITE` request would then fail if the targeted database is a standard database.

Example 370. DROP COMPOSITE DATABASE

Query

```
DROP COMPOSITE DATABASE inventory
```

0 rows, System updates: 1

To ensure the database to be dropped is standard and not composite, the user first needs to check the `type` column of `SHOW DATABASES` manually.

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 371. CREATE DATABASE

Query

```
CREATE DATABASE slow WAIT 5 SECONDS
```

Table 538. 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.

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.

Database alias management

This section explains how to use Cypher to manage database aliases in Neo4j.

There are two kinds of database aliases: local and remote. A local database alias can only target a database within the same DBMS. A remote database alias may target a database from another Neo4j DBMS. When a query is run against a database 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. Both local and remote database aliases can be created as part of a [composite database](#).

A local database alias can be used in all other Cypher commands in place of the target database. Please note that the local database alias will be resolved while executing the command. Privileges are defined on the database, and not the local database alias.

A remote database 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 database aliases require configuration to safely connect to the remote target, which is described in [Connecting remote databases](#). It is not possible to impersonate a user on the remote database or to execute an administration command on the remote database via a remote database 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 database alias management commands is as follows:



The syntax descriptions use [the style](#) from access control.

Table 539. Alias management command syntax

Command	Syntax
Show Database Alias	<pre>SHOW ALIAS[ES] [name] FOR DATABASE[S] [WHERE expression]</pre> <pre>SHOW ALIAS[ES] [name] FOR DATABASE[S] YIELD { * field[, ...] } [ORDER BY field[, ...]] [SKIP n] [LIMIT n] [WHERE expression] [RETURN field[, ...] [ORDER BY field[, ...]] [SKIP n] [LIMIT n]]</pre>
	<p>Lists both local and remote database aliases, optionally filtered on the alias name.</p>
Create Local Alias	<pre>CREATE ALIAS name [IF NOT EXISTS] FOR DATABASE targetName [PROPERTIES "{" key: value[, ...] "}"]</pre> <pre>CREATE OR REPLACE ALIAS name FOR DATABASE targetName [PROPERTIES "{" key: value[, ...] "}"]</pre>

Command	Syntax
Create Remote Alias	<pre>CREATE ALIAS name [IF NOT EXISTS] FOR DATABASE targetName AT 'url' USER username PASSWORD 'password' [DRIVER "{" setting: value[, ...] "}"] [PROPERTIES "{" key: value[, ...] "}"]</pre>
	<pre>CREATE OR REPLACE ALIAS name FOR DATABASE targetName AT 'url' USER username PASSWORD 'password' [DRIVER {" setting: value[, ...] "}"] [PROPERTIES {" key: value[, ...] "}"}}</pre>
Alter Local Alias	<pre>ALTER ALIAS name [IF EXISTS] SET DATABASE [TARGET targetName] [PROPERTIES {" key: value[, ...] "}"}</pre>
Alter Remote Alias	<pre>ALTER ALIAS name [IF EXISTS] SET DATABASE [TARGET targetName AT 'url'] [USER username] [PASSWORD 'password'] [DRIVER {" setting: value[, ...] "}"] [PROPERTIES {" key: value[, ...] "}"}}</pre>
Drop Alias	<pre>DROP ALIAS name [IF EXISTS] FOR DATABASE</pre> <p>Drop either a local or remote database alias.</p>

This is the list of the allowed driver settings for remote database aliases.

Table 540. `ssl_enforced`

Description	SSL for remote database 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 541. `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	dbms.routing.driver.connection.connect_timeout
---------------	--

Table 542. 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	<p>Duration.</p> <p>Zero and negative values result in lifetime not being checked.</p>
Default value	dbms.routing.driver.connection.max_lifetime

Table 543. 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	<p>Duration.</p> <p>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.</p>
Default value	dbms.routing.driver.connection.pool.acquisition_timeout

Table 544. 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	<p>Integer.</p> <p>Negative values are allowed and result in unlimited pool. Value of 0 is not allowed.</p>
Default value	dbms.routing.driver.connection.pool.max_size

Table 545. logging_level

Description	Sets level for driver internal logging.
Valid values	org.neo4j.logging.Level. One of DEBUG, INFO, WARN, ERROR, or NONE.
Default value	dbms.routing.driver.logging.level



If transaction modifies a database 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 fully qualified name of the database alias. Default output
<code>database</code>	The name of the target database. Default output
<code>location</code>	The location of the database, either local or remote. Default output
<code>url</code>	Target location or <code>null</code> if the target is local. Default output
<code>user</code>	User connecting to the remote database or <code>null</code> if the target database is local. Default output
<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 driver settings allowed for remote database aliases.
<code>properties</code>	Any properties set on the database alias.

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 372. Show all aliases for a database

A summary of all available database aliases can be displayed using the command `SHOW ALIASES FOR DATABASE`.

Query

```
SHOW ALIASES FOR DATABASE
```

Table 546. Result

name	database	location	url	user
"films"	"movies"	"local"	<null>	<null>
"library.romance"	romance-books"	"remote"	"neo4j+s://location: 7687"	"alice"
"library.sci-fi"	sci-fi-books"	"local"	<null>	<null>
"motion pictures"	"movies"	"local"	<null>	<null>
"movie scripts"	"scripts"	"remote"	"neo4j+s://location: 7687"	"alice"

Rows: 5

Example 373. Show specific aliases for databases

To list just one database alias, the `SHOW ALIASES` command takes an alias name;

Query

```
SHOW ALIAS films FOR DATABASES
```

Table 547. Result

name	database	location	url	user
"films"	"movies"	"local"	<null>	<null>
Rows: 1				

Query

```
SHOW ALIAS library.romance FOR DATABASES
```

Table 548. Result

name	database	location	url	user
"library.romance"	romance-books"	"remote"	"neo4j+s://location: 7687"	"alice"
Rows: 1				

Example 374. Show detailed aliases information for a database

Query

```
SHOW ALIASES FOR DATABASE YIELD *
```

Table 549. Result

name	database	location	url	user	driver	properties
"films"	"movies"	"local"	<null>	<null>	<null>	{}
"library.romance"	"romance-books"	"remote"	"neo4j+s://location:7687"	"alice"	{}	{}
"library.sci-fi"	"sci-fi-books"	"local"	<null>	<null>	<null>	{}
"motion pictures"	"movies"	"local"	<null>	<null>	<null>	{"namecontains space":true}
"movie scripts"	"scripts"	"remote"	"neo4j+s://location:7687"	"alice"	+{"connection_pool_idle_test":PT2M,"connection_pool_max_size":10,"logging_connection_pool_idle_test":PT2M,"connection_pool_max_size":10,"logging_level":"INFO","ssl_enforced":true,"connection_pool_acquisition_time_out":PT1M,"connection_timeout":PT5S,"connection_max_lifetime":PT1H}	{}

Rows: 5

Example 375. Show count of aliases for a 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 550. Result

count
5
Rows: 1

Example 376. Show filtered aliases information for a 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 551. Result

name	url	database
"motion pictures"	<null>	"movies"
"library.romance"	"neo4j+s://location:7687"	"romance-books"
"movie scripts"	"neo4j+s://location:7687"	"scripts"

Rows: 3

Creating database aliases Enterprise edition

Database aliases can be created using `CREATE ALIAS`.

The required privileges are described [here](#).

Table 552. Create alias command syntax

Syntax	Comment
<pre>CREATE [OR REPLACE] ALIAS [compositeDatabaseName.]aliasName [IF NOT EXISTS] FOR DATABASE targetName [PROPERTIES "{" key: value[, ...] "}"]</pre>	Create a local alias.
<pre>CREATE [OR REPLACE] ALIAS [compositeDatabaseName.]aliasName [IF NOT EXISTS] FOR DATABASE targetName AT 'url' USER username PASSWORD 'password' [DRIVER "{" setting: value[, ...] "}"] [PROPERTIES "{" key: value[, ...] "}"]</pre>	Create a remote database alias.

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.



Database alias names are subject to the [standard Cypher restrictions on valid identifiers](#).

The following naming rules apply:

- A name is a valid identifier.
- 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 377. Creating aliases for local databases

Query

```
CREATE ALIAS `northwind` FOR DATABASE `northwind-graph-2021`
```

System updates: 1
Rows: 0

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 553. Result

name	type	aliases	access	addresses	role	writer	requestedStatus	currentStatus	statusMessage	default	home	constituents
"northwind-graph-2021"	+"standard"	["northwind"]	"read-write"	"localhost:7687"	"primary"	"true"	"online"	"online"	"	false	false	[]

Rows: 1

Query

```
SHOW ALIAS `northwind` FOR DATABASE
```

Table 554. Result

name	database	location	url	user
"northwind"	"northwind-graph-2021"	"local"	<null>	<null>

Rows: 1

Example 378. Setting properties for local database aliases

Local database aliases can also be given properties.

Query

```
CREATE ALIAS `northwind-2022`  
FOR DATABASE `northwind-graph-2022`  
PROPERTIES { newestNorthwind: true, index: 3 }
```

```
System updates: 1  
Rows: 0
```

The properties are then shown in the `SHOW ALIASES FOR DATABASE YIELD ...` command.

Query

```
SHOW ALIAS `northwind-2022` FOR DATABASE YIELD name, properties
```

Table 555. Result

name	properties
"northwind-2022"	{"index":3,"newestnorthwind":true}
Rows: 1	

Example 379. Creating database aliases with the same name as an existing alias

Adding a local database 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`
```

```
(no changes, no records)
```

Example 380. Creating or replacing database aliases

It is also possible to replace a database 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 the following two queries consecutively:

Query

```
DROP ALIAS `northwind` IF EXISTS FOR DATABASE
```

Query

```
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 database aliases also allows `IF NOT EXISTS` and `OR REPLACE` clauses. Both check for any remote or local database aliases.

Example 381. Creating remote database aliases

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
```

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 ALIAS `remote-northwind`
FOR DATABASE
```

Table 556. Result

name	database	location	url	user
"remote-northwind"	"northwind-graph-2020"	"remote"	"neo4j+s://location:7687"	"alice"

Rows: 1

Example 382. Creating remote database aliases with driver settings

It is possible to override the default driver settings per database 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

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 ALIAS `remote-with-driver-settings` FOR DATABASE YIELD *
```

Table 557. Result

name	database	location	url	user	driver	properties
"remote-with-driver-settings"	"northwind-graph-2020"	"remote"	"neo4j+s://location:7687"	"alice"	{connection_pool_max_size -> 10, connection_timeout -> PT1M}	{}

Rows: 1

Example 383. Setting properties for remote database aliases

Just as the local database aliases, the remote database aliases can be given properties.

Query

```
CREATE ALIAS 'remote-northwind-2021' FOR DATABASE 'northwind-graph-2021' AT 'neo4j+s://location:7687'  
USER alice PASSWORD 'password'  
PROPERTIES { newestNorthwind: false, index: 6 }
```

System updates: 1
Rows: 0

The properties are then shown in the `SHOW ALIASES FOR DATABASE YIELD ...` command.

Query

```
SHOW ALIAS 'remote-northwind-2021' FOR DATABASE YIELD name, properties
```

Table 558. Result

name	properties
"remote-northwind-2021"	{"index":6,"newestnorthwind":false}
Rows: 1	

Create database aliases in composite databases Enterprise edition

Both local and remote database aliases can be part of a [composite database](#).

Create a database alias in a composite database by giving the name of the composite database as namespace for the alias.

Example 384. Creating aliases in composite databases

Query

```
CREATE ALIAS garden.flowers  
FOR DATABASE `perennial-flowers`
```

```
System updates: 1  
Rows: 0
```

Query

```
CREATE ALIAS garden.trees  
FOR DATABASE trees AT 'neo4j+s://location:7687'  
USER alice PASSWORD 'password'
```

```
System updates: 1  
Rows: 0
```

When a database alias has been created in a composite database, it will show up in the `constituents` column provided by the command `SHOW DATABASES` and in the `SHOW ALIASES FOR DATABASE` command.

Query

```
SHOW DATABASE garden YIELD name, type, constituents
```

Table 559. Result

name	type	constituents
"garden"	"composite"	["garden.flowers", "garden.trees"]
Rows: 1		

Query

```
SHOW ALIASES FOR DATABASE WHERE name STARTS WITH 'garden'
```

Table 560. Result

name	database	location	url	user
"garden.flowers"	"perennial-flowers"	"local"	<null>	<null>
"garden.trees"	"trees"	"remote"	"neo4j+s://location:7687"	"alice"
Rows: 1				

Example 385. Aliases pointing to composite databases

Database aliases cannot point to a composite database.

Query

```
CREATE ALIAS yard FOR DATABASE garden
```

Error message

```
Failed to create the specified database alias 'yard': Database 'garden' is composite.
```

Altering database aliases

Database aliases can be altered using `ALTER ALIAS` to change its database target, properties, url, user credentials, or driver settings. The required privileges are described [here](#). Only the clauses used will be altered.



Local database aliases cannot be altered to remote aliases, or vice versa.

Table 561. Alter alias command syntax

Syntax	Comment
<pre>ALTER ALIAS [compositeDatabaseName.]aliasName [IF EXISTS] SET DATABASE [TARGET targetName] [PROPERTIES "{" key: value[, ...] "}"]</pre>	Modify database target of a local alias. The clauses can be applied in any order, while at least one clause needs to be set.
<pre>ALTER ALIAS [compositeDatabaseName.]aliasName [IF EXISTS] SET DATABASE [TARGET targetName AT 'url'] [USER username] [PASSWORD 'password'] [DRIVER "{" setting: value[, ...] "}"] [PROPERTIES "{" key: value[, ...] "}"]</pre>	Modify a remote alias. The clauses can be applied in any order, while at least one clause needs to be set.

Example 386. Altering local database aliases

Example of altering a local database alias target.

Query

```
ALTER ALIAS 'northwind'  
SET DATABASE TARGET 'northwind-graph-2021'
```

```
System updates: 1  
Rows: 0
```

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-graph-2021'
```

Table 562. Result

name	type	aliases	access	addresses	role	writer	requestedStatus	currentStatus	statusMessage	default	home	constituents
"northwind-graph-2021"	"standalone"	["northwind"]	"read-write"	"localhost:7687"	"primary"	"true"	"online"	"online"	"	false	false	[]

Rows: 1

Example 387. Altering remote database aliases

Example of altering a remote database alias target.

Query

```
ALTER ALIAS 'remote-northwind' SET DATABASE  
TARGET 'northwind-graph-2020' AT "neo4j+://other-location:7687"
```

```
System updates: 1  
Rows: 0
```

Example 388. Altering remote credentials and driver settings for remote database aliases

Example of altering a remote database 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 389. Removing custom driver settings from remote database aliases

Example of altering a remote database alias to remove all custom driver settings.

Query

```
ALTER ALIAS 'movie scripts' SET DATABASE
DRIVER {}
```

```
System updates: 1
Rows: 0
```

Example 390. Altering properties for local and remote database aliases

Examples of altering local and remote database alias properties.

Query

```
ALTER ALIAS 'motion pictures' SET DATABASE PROPERTIES { nameContainsSpace: true, moreInfo: 'no, not  
really' }
```

```
System updates: 1  
Rows: 0
```

Query

```
ALTER ALIAS 'movie scripts' SET DATABASE PROPERTIES { nameContainsSpace: true }
```

```
System updates: 1  
Rows: 0
```

Example 391. Altering local and remote aliases in composite databases

Examples of altering local and remote database alias in composite databases.

Query

```
ALTER ALIAS garden.flowers SET DATABASE PROPERTIES { perennial: true }
```

```
System updates: 1  
Rows: 0
```

Query

```
ALTER ALIAS garden.trees SET DATABASE TARGET updatedTrees AT 'neo4j+s://location:7687' PROPERTIES {  
treeVersion: 2 }
```

```
System updates: 1  
Rows: 0
```

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',  
'motion pictures', 'garden.flowers', 'garden.trees']
```

Table 563. Result

name	database	location	url	user	driver	properties
"garden.flowers"	"perennial-flowers"	"local"	<null>	<null>	<null>	{"perennial":true}
"garden.trees"	"updatedtrees"	"remote"	"neo4j+s://location:7687"	"alice"	{}	{"treeversion":2}
"motion pictures"	"movies"	"local"	<null>	<null>	<null>	{"namecontains space":true,"moreinfo":"no, not really"}
"movie scripts"	"scripts"	"remote"	"neo4j+s://location:7687"	"alice"	{}	{"namecontains space":true}
"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: 7

Example 392. Using IF EXISTS when altering database aliases

The `ALTER ALIAS` command is optionally idempotent, with the default behavior to fail with an error if the database 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`
```

```
(no changes, no records)
```

Deleting database aliases Enterprise edition

Both local and remote database aliases can be deleted using the `DROP ALIAS` command. The required privileges are described [here](#).

Example 393. Deleting local database aliases

Delete a local database alias.

Query

```
DROP ALIAS `northwind` FOR DATABASE
```

```
System updates: 1  
Rows: 0
```

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 564. Result

name	type	aliases	access	addresses	role	writer	requestedStatus	currentStatus	statusMessage	default	home	constituents
"northwind-graph-2021"	"standalone"	[]	"read-write"	"localhost:7687"	"primary"	"true"	"online"	"online"	"	false	false	[]

Rows: 1

Example 394. Deleting remote database aliases

Delete a remote database alias.

Query

```
DROP ALIAS `remote-northwind` FOR DATABASE
```

```
System updates: 1  
Rows: 0
```

Example 395. Deleting aliases in composite databases

Delete an alias in a composite database.

Query

```
DROP ALIAS garden.flowers FOR DATABASE
```

```
System updates: 1  
Rows: 0
```

When a database alias has been deleted, it will no longer show up in the `SHOW ALIASES FOR DATABASE` command.

Query

```
SHOW ALIASES FOR DATABASE
```

Table 565. Result

name	database	location	url	user
"films"	"movies"	"local"	<null>	<null>
"garden.trees"	"updatedtrees"	"local"	<null>	<null>
"library.romance"	"romance-books"	"remote"	"neo4j+s://location:7687"	"alice"
"library.sci-fi"	"sci-fi-books"	"local"	<null>	<null>
"motion pictures"	"movies"	"local"	<null>	<null>
"movie scripts"	"scripts"	"remote"	"neo4j+s://location:7687"	"alice"
"northwind-2022"	"northwind-graph-2022"	"local"	<null>	<null>
"remote-northwind-2021"	"northwind-graph-2021"	"remote"	"neo4j+s://location:7687"	"alice"
"remote-with-driver-settings"	"northwind-graph-2020"	"remote"	"neo4j+s://location:7687"	"bob"
Rows: 9				

Example 396. Using IF EXISTS when deleting database aliases

The `DROP ALIAS` command is optionally idempotent, with the default behavior to fail with an error if the database 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
```

```
(no changes, no records)
```

Access control

This section explains how to manage Neo4j role-based access control and fine-grained security.

Neo4j has a complex security model stored in the system graph, which is maintained on a special database called the `system` database. All administrative commands need to be executed against the `system` database. When connected to the DBMS over `bolt`, administrative commands are automatically routed to the `system` database. For more information on how to manage multiple databases, refer to the section on [administering databases](#).

The concept of role-based access control was introduced in Neo4j 3.1. Since then, it has been possible to create users and assign them to roles to control whether users can read, write and administer the database. In Neo4j 4.0 this model was enhanced significantly with the addition of privileges, which are the underlying access-control rules by which the users rights are defined.

The original built-in roles still exist with almost the exact same access rights, but they are no-longer statically defined (see [Built-in roles](#)). Instead, they are defined in terms of their underlying privileges, and they can be modified by adding 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.

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 used to describe all versions of a command. These summaries use some special characters to indicate such variations.

Table 566. Special characters in syntax summaries

Character	Meaning	Example
	Used to indicate alternative parts of a command (i.e. <code>or</code>). Needs to be part of a grouping.	If the syntax needs to specify either a name or <code>*</code> , this can be indicated with <code>* name</code> .
{ and }	Used to group parts of the command. Commonly found together with .	In order to use the <code>or</code> in the syntax summary, it needs to be in a group: <code>{* name}</code> .

Character	Meaning	Example
[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 S is optional with GRAPH[S].
...	Repeated pattern. Related to the command part immediately before this is repeated.	A comma separated list of names would be name[, ...].
"	When a special character is part of the syntax itself, we surround it with " to indicate this.	To include { in the syntax use "{" {*} name } "}". In this case, you will get either {*} or { name }.

The special characters in the table above are the only ones that need to be escaped using " in the syntax summaries.

Here is an example that uses all the special characters. It grants 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
```

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 management command syntax



The syntax descriptions use the style from access control.

Command	SHOW CURRENT USER
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 Listing current user.</p>
Required privilege	None

Command	SHOW USERS
Syntax	<pre>SHOW USER[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 all users.</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 Listing users.</p>
Required privilege	<pre>GRANT SHOW USER</pre> <p>(see DBMS USER MANAGEMENT privileges)</p>

Command	SHOW USER PRIVILEGES
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 RETURN clause, the YIELD clause is mandatory and must not be omitted.</p> <p>For more information, see Listing privileges.</p>
Required privilege	<div style="border: 1px solid #ccc; padding: 5px; margin-bottom: 10px;"> <pre>GRANT SHOW PRIVILEGE</pre> </div> <p>(see DBMS PRIVILEGE MANAGEMENT privileges)</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <pre>GRANT SHOW USER</pre> </div> <p>(see DBMS USER MANAGEMENT privileges)</p>

Command	CREATE USER
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 Creating users.</p>
Required privilege	<div style="border: 1px solid #ccc; padding: 5px; margin-bottom: 10px;"> <pre>GRANT CREATE USER</pre> </div> <p>(see DBMS USER MANAGEMENT privileges)</p>

Command	CREATE OR REPLACE USER
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 Creating users.</p>

Required privilege	<p>GRANT CREATE USER</p> <p>(see DBMS USER MANAGEMENT privileges)</p> <p>GRANT DROP USER</p> <p>(see DBMS USER MANAGEMENT privileges)</p>
--------------------	---

Command	RENAME USER
Syntax	RENAME USER name [IF EXISTS] TO otherName
Description	<p>Changes the name of a user.</p> <p>For more information, see Renaming users.</p>
Required privilege	<p>GRANT RENAME USER</p> <p>(see DBMS USER MANAGEMENT privileges)</p>

Command	ALTER USER
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 SET or REMOVE clause is required. SET and REMOVE clauses cannot be combined in the same command.</p> <p>For more information, see Modifying users.</p>

Required privilege	GRANT SET PASSWORD
	[source, privilege, role="noheader"]
	GRANT SET USER STATUS
	GRANT SET USER HOME DATABASE

(see [DBMS USER MANAGEMENT privileges](#))

Command	ALTER CURRENT USER SET PASSWORD
Syntax	ALTER CURRENT USER SET PASSWORD FROM 'oldPassword' TO 'newPassword'
Description	<p>Changes the current user's password.</p> <p>For more information, see Changing the current user's password.</p>
Required privilege	None

Command	DROP USER
Syntax	DROP USER name [IF EXISTS]
Description	<p>Removes an existing user.</p> <p>For more information, see Delete users.</p>
Required privilege	GRANT DROP USER (see DBMS USER MANAGEMENT privileges)



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 567. 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).	✗	✓

Column	Description	Community Edition	Enterprise Edition
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 568. 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 set-initial-password`, otherwise it is necessary to change the password after the first login.

Example 397. Show user

This example shows how to:

- Reorder the columns using a `YIELD` clause.
- Filter the results using a `WHERE` clause.

```
SHOW USER YIELD user, suspended, passwordChangeRequired, roles, home
WHERE user = 'jake'
```

Example 398. 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.
- The default Neo4j password length is at least 8 characters.
- 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>, <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 399. 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 'abcd1234' CHANGE REQUIRED
SET STATUS SUSPENDED
SET HOME DATABASE anotherDb
```

Example 400. 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 401. Create user if not exists

```
CREATE USER jake IF NOT EXISTS
SET PLAINTEXT PASSWORD 'abcd1234'
```

The `CREATE OR REPLACE USER` command will result in any existing user being deleted and a new one created.

Example 402. Create or replace user

```
CREATE OR REPLACE USER jake  
SET PLAINTEXT PASSWORD 'abcd1234'
```

This is equivalent to running `DROP USER jake IF EXISTS` followed by `CREATE USER jake SET PASSWORD 'abcd1234'`.



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 569. 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>, <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 'abcd1234' 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 570. 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 'abcd1234'
```

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 'password1' TO 'password2'
```



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 571. 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



The syntax descriptions use the style from access control.

Command	<code>SHOW ROLES</code>
Syntax	<pre>SHOW [ALL POPULATED] ROLE[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 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 Listing roles.</p>
Required privilege	<pre>GRANT SHOW ROLE</pre> <p>(see DBMS ROLE MANAGEMENT privileges).</p>

Command	<code>SHOW ROLES WITH USERS</code>
Syntax	<pre>SHOW [ALL POPULATED] ROLE[S] WITH USER[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 roles and users assigned to them.</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 Listing roles.</p>
Required privilege	<p><code>GRANT SHOW ROLE</code></p> <p>(see DBMS ROLE MANAGEMENT privileges)</p> <p><code>GRANT SHOW USER</code></p> <p>(see DBMS USER MANAGEMENT privileges)</p>

Command	<code>SHOW ROLE PRIVILEGES</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	<p>Lists the privileges granted to the specified 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 Listing privileges.</p>
Required privilege	<p><code>GRANT SHOW PRIVILEGE</code></p> <p>(see DBMS PRIVILEGE MANAGEMENT privileges)</p>

Command	<code>CREATE ROLE</code>
Syntax	<pre>CREATE ROLE name [IF NOT EXISTS] [AS COPY OF otherName]</pre>
Description	<p>Creates a new role.</p> <p>For more information, see Creating roles.</p>
Required privilege	<p><code>GRANT CREATE ROLE</code></p> <p>(see DBMS ROLE MANAGEMENT privileges)</p>

Command	CREATE OR REPLACE ROLE
Syntax	CREATE OR REPLACE ROLE name [AS COPY OF otherName]
Description	<p>Creates a new role, or if a role with the same name exists, replace it.</p> <p>For more information, see Creating roles.</p>
Required privilege	<p>GRANT CREATE ROLE</p> <p>GRANT DROP ROLE</p> <p>(see DBMS ROLE MANAGEMENT privileges)</p>

Command	RENAME ROLE
Syntax	RENAME ROLE name [IF EXISTS] TO otherName
Description	<p>Changes the name of a role.</p> <p>For more information, see Renaming roles.</p>
Required privilege	<p>GRANT RENAME ROLE</p> <p>(see DBMS ROLE MANAGEMENT privileges)</p>

Command	DROP ROLE
Syntax	DROP ROLE name [IF EXISTS]
Description	<p>Removes a role.</p> <p>For more information, see Deleting roles.</p>

Command	GRANT ROLE TO
Syntax	GRANT ROLE[S] name[, ...] TO user[, ...]

Description	Assigns roles to users. For more information, see Assigning roles to users .
Required privilege	GRANT ASSIGN ROLE (see DBMS ROLE MANAGEMENT privileges)

Command	REVOKE ROLE
Syntax	REVOKE ROLE[S] name[, ...] FROM user[, ...]
Description	Removes roles from users. For more information, see Revoking roles from users .
Required privilege	GRANT REMOVE ROLE (see DBMS ROLE MANAGEMENT privileges)

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 572. 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 573. 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 574. 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 403. Copy a role

```
CREATE ROLE mysecondrole AS COPY OF myrole
```

Created roles will appear on the list provided by `SHOW ROLES`.

Example 404. List roles

```
SHOW ROLES
```

Table 575. 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 405. 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 406. 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 576. 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 ROLES
```

Table 577. 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>

Rows: 5

It is possible to assign multiple roles to multiple users in one command:

```
GRANT ROLES role1, role2 TO user1, user2, user3
```

```
SHOW ROLES
```

Table 578. 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 ROLES
```

Table 579. 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 580. 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](#).



The syntax descriptions use [the style](#) from access control.

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.
- mutability:
 - [IMMUTABLE](#) can optionally be specified when performing a [GRANT](#) or [DENY](#) to indicate that the privilege cannot be subsequently removed unless auth is disabled. Auth must also be disabled in order to [GRANT](#) or [DENY](#) an immutable privilege. Contrastingly, when [IMMUTABLE](#) is specified in conjunction with a [REVOKE](#) command, it will act as a filter and only remove matching immutable privileges. See also [immutable privileges](#).
- graph-privilege:
 - Can be either a [read privilege](#) or [write privilege](#).
- name:
 - The graph or graphs to associate the privilege with. Because in Neo4j 5 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 581. General grant ON GRAPH privilege syntax

Command	GRANT ... ON ... TO ...
Syntax	GRANT [IMMUTABLE] graph-privilege ON { HOME GRAPH GRAPH[S] { * name[, ...] } } [entity] TO role[, ...]
Description	Grants a privilege to one or multiple roles.

Table 582. General deny ON GRAPH privilege syntax

Command	DENY ... ON ... TO ...
Syntax	DENY [IMMUTABLE] graph-privilege ON { HOME GRAPH GRAPH[S] { * name[, ...] } } [entity] TO role[, ...]
Description	Denies a privilege to one or multiple roles.

Table 583. General revoke ON GRAPH privilege syntax

Command	REVOKE GRANT ... ON ... FROM ...
Syntax	REVOKE [IMMUTABLE] GRANT graph-privilege ON { HOME GRAPH GRAPH[S] { * name[, ...] } } [entity] FROM role[, ...]
Description	Revokes a granted privilege from one or multiple roles.

Table 584. General revoke ON GRAPH privilege syntax

Command	<code>REVOKE DENY ... ON ... FROM ...</code>
Syntax	<code>REVOKE [IMMUTABLE] DENY graph-privilege ON { HOME GRAPH GRAPH[S] { * name[, ...] } } [entity] FROM role[, ...]</code>
Description	Revokes a denied privilege from one or multiple roles.

Table 585. General revoke ON GRAPH privilege syntax

Command	<code>REVOKE ... ON ... FROM ...</code>
Syntax	<code>REVOKE [IMMUTABLE] graph-privilege ON { HOME GRAPH GRAPH[S] { * name[, ...] } } [entity] FROM role[, ...]</code>
Description	Revokes a granted or denied privilege from one or multiple roles.



`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 586. 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 587. 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 588. 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.
- `immutable`: whether or not the privilege is immutable.

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 589. Result

access	action	resource	graph	segment	role	immutable
"GRANTED"	"execute"	"database"	"*"	"FUNCTION(*)"	"PUBLIC"	false
"GRANTED"	"execute"	"database"	"*"	"PROCEDURE(*)"	"PUBLIC"	false
"GRANTED"	"access"	"database"	"DEFAULT"	"database"	"PUBLIC"	false

access	action	resource	graph	segment	role	immutable
"GRANTED"	"match"	"all_properties"	"*"	"NODE(*)"	"admin"	false
"GRANTED"	"write"	"graph"	"*"	"NODE(*)"	"admin"	false
"GRANTED"	"match"	"all_properties"	"*"	"RELATIONSHIP(*)"	"admin"	false
"GRANTED"	"write"	"graph"	"*"	"RELATIONSHIP(*)"	"admin"	false
"GRANTED"	"transaction_management"	"database"	"*"	"USER(*)"	"admin"	false
"GRANTED"	"access"	"database"	"*"	"database"	"admin"	false
"GRANTED"	"constraint"	"database"	"*"	"database"	"admin"	false
"GRANTED"	"dbms_actions"	"database"	"*"	"database"	"admin"	false
"GRANTED"	"index"	"database"	"*"	"database"	"admin"	false
"GRANTED"	"start_database"	"database"	"*"	"database"	"admin"	false
"GRANTED"	"stop_database"	"database"	"*"	"database"	"admin"	false
"GRANTED"	"token"	"database"	"*"	"database"	"admin"	false
"GRANTED"	"match"	"all_properties"	"*"	"NODE(*)"	"architect"	false
"GRANTED"	"write"	"graph"	"*"	"NODE(*)"	"architect"	false
"GRANTED"	"match"	"all_properties"	"*"	"RELATIONSHIP(*)"	"architect"	false
"GRANTED"	"write"	"graph"	"*"	"RELATIONSHIP(*)"	"architect"	false
"GRANTED"	"access"	"database"	"*"	"database"	"architect"	false
"GRANTED"	"constraint"	"database"	"*"	"database"	"architect"	false
"GRANTED"	"index"	"database"	"*"	"database"	"architect"	false
"GRANTED"	"token"	"database"	"*"	"database"	"architect"	false
"GRANTED"	"match"	"all_properties"	"*"	"NODE(*)"	"editor"	false
"GRANTED"	"write"	"graph"	"*"	"NODE(*)"	"editor"	false
"GRANTED"	"match"	"all_properties"	"*"	"RELATIONSHIP(*)"	"editor"	false
"GRANTED"	"write"	"graph"	"*"	"RELATIONSHIP(*)"	"editor"	false
"GRANTED"	"access"	"database"	"*"	"database"	"editor"	false
"DENIED"	"access"	"database"	"neo4j"	"database"	"noAccessUsers"	false
"GRANTED"	"match"	"all_properties"	"*"	"NODE(*)"	"publisher"	false
"GRANTED"	"write"	"graph"	"*"	"NODE(*)"	"publisher"	false
"GRANTED"	"match"	"all_properties"	"*"	"RELATIONSHIP(*)"	"publisher"	false

access	action	resource	graph	segment	role	immutable
"GRANTED"	"write"	"graph"	"*"	"RELATIONSHIP(*)"	"publisher"	false
"GRANTED"	"access"	"database"	"*"	"database"	"publisher"	false
"GRANTED"	"token"	"database"	"*"	"database"	"publisher"	false
"GRANTED"	"match"	"all_properties"	"*"	"NODE(*)"	"reader"	false
"GRANTED"	"match"	"all_properties"	"*"	"RELATIONSHIP(*)"	"reader"	false
"GRANTED"	"access"	"database"	"*"	"database"	"reader"	false
"GRANTED"	"access"	"database"	"neo4j"	"database"	"regularUsers"	false

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 590. 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 591. 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 592. Result

role	access	privileges
"PUBLIC"	"GRANTED"	[["*", "database", "FUNCTION(*)", "execute"], ["*", "database", "PROCEDURE(*)", "execute"], ["DEFAULT", "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"]]

role	access	privileges
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 593. Result

access	action	graph	resource	role	segment	immutable
"GRANTED"	"match"	"*"	"all_properties"	"admin"	"RELATIONSHIP(*)"	false
"GRANTED"	"write"	"*"	"graph"	"admin"	"RELATIONSHIP(*)"	false
"GRANTED"	"transaction_management"	"*"	"database"	"admin"	"USER(*)"	false
"GRANTED"	"access"	"*"	"database"	"admin"	"database"	false
"GRANTED"	"constraint"	"*"	"database"	"admin"	"database"	false

Available privileges can also be displayed as Cypher commands by adding `AS COMMAND[S]`:

```
SHOW PRIVILEGES AS COMMANDS
```

Table 594. 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`"

command

```
"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`"  
"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 595. 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`"

command

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 596. 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`"

command

```
"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 597. Result

access	action	graph	resource	role	segment	immutable
"GRANTED"	"access"	"database"	"neo4j"	"database"	"regularUsers"	false

```
SHOW ROLES regularUsers, noAccessUsers PRIVILEGES
```

Lists all privileges for roles `regularUsers` and `noAccessUsers`.

Table 598. Result

access	action	graph	resource	role	segment	immutable
"DENIED"	"access"	"database"	"neo4j"	"database"	"noAccessUsers"	false
"GRANTED"	"access"	"database"	"neo4j"	"database"	"regularUsers"	false

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 599. 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`"
```

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 600. 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 601. 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 602. Result

access	action	resource	graph	resource	role	segment	immutable
"GRANTED"	"execute"	"database"	"*"	"FUNCTION(*)" "	"PUBLIC"	"jake"	false
"GRANTED"	"execute"	"database"	"*"	"PROCEDURE(*)"	"PUBLIC"	"jake"	false
"GRANTED"	"access"	"database"	"DEFAULT"	"database"	"PUBLIC"	"jake"	false
"GRANTED"	"access"	"database"	"neo4j"	"database"	"regularUsers"	"jake"	false

```
SHOW USERS jake, joe PRIVILEGES
```

Lists all privileges for users `jake` and `joe`.

Table 603. Result

access	action	resource	graph	resource	role	segment	immutable
"GRANTED"	"execute"	"database"	"*"	"FUNCTION(*)" "	"PUBLIC"	"jake"	false
"GRANTED"	"execute"	"database"	"*"	"PROCEDURE(*)"	"PUBLIC"	"jake"	false
"GRANTED"	"access"	"database"	"DEFAULT"	"database"	"PUBLIC"	"jake"	false
"GRANTED"	"access"	"database"	"neo4j"	"database"	"regularUsers"	"jake"	false
"GRANTED"	"execute"	"database"	"*"	"FUNCTION(*)" "	"PUBLIC"	"joe"	false
"GRANTED"	"execute"	"database"	"*"	"PROCEDURE(*)"	"PUBLIC"	"joe"	false

access	action	resource	graph	resource	role	segment	immutable
"GRANTED"	"access"	"database"	"DEFAULT"	"database"	"PUBLIC"	"joe"	false
"DENIED"	"access"	"database"	"neo4j"	"database"	"noAccessUsers"	"joe"	false

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 604. Result

command
<code>"GRANT ACCESS ON DATABASE neo4j TO \$role"</code>
<code>"GRANT ACCESS ON HOME DATABASE TO \$role"</code>
<code>"GRANT EXECUTE FUNCTION * ON DBMS TO \$role"</code>
<code>"GRANT EXECUTE PROCEDURE * ON DBMS TO \$role"</code>
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 605. Result

command
<code>"REVOKE GRANT EXECUTE FUNCTION * ON DBMS FROM \$role"</code>
<code>"REVOKE GRANT EXECUTE PROCEDURE * ON DBMS FROM \$role"</code>
Rows: 2

Revoking privileges

Enterprise edition

Privileges that were granted or denied earlier can be revoked using the `REVOKE` command:

```
REVOKE
  [ IMMUTABLE ]
  [ 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
```

Adding `IMMUTABLE` explicitly specifies that only immutable privileges should be removed. Omitting it specifies that both immutable and regular privileges should be removed.

Managing servers

Servers can be added 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.

Server management command syntax



The syntax descriptions use the style from access control.

Command	<code>ENABLE SERVER</code>
Syntax	<pre>ENABLE SERVER 'serverId' [OPTIONS "{" option: value[,...] "}"]</pre>
Description	Adds a server that has been discovered to the cluster. For more information see Enabling servers .
Required privilege	<code>GRANT SERVER MANAGEMENT</code> (see SERVER MANAGEMENT privileges)

Command	<code>ALTER SERVER</code>
Syntax	<code>ALTER SERVER 'name' SET OPTIONS "{" option: value[,...] "}"</code>
Description	Changes the constraints for a server. For more information see Modifying servers .
Required privilege	<code>GRANT SERVER MANAGEMENT</code> (see SERVER MANAGEMENT privileges)

Command	<code>RENAME SERVER</code>
Syntax	<code>RENAME SERVER 'name' TO 'newName'</code>
Description	Changes the name of a server. For more information see Renaming servers .
Required privilege	<code>GRANT SERVER MANAGEMENT</code> (see SERVER MANAGEMENT privileges)

Command	<code>REALLOCATE DATABASES</code>
Syntax	<code>[DRYRUN] REALLOCATE DATABASE[S]</code>
Description	Re-balances databases among the servers in the cluster. For more information see Reallocate databases .
Required privilege	<code>GRANT SERVER MANAGEMENT</code> (see SERVER MANAGEMENT privileges)

Command	<code>DEALLOCATE DATABASES</code>
Syntax	<code>[DRYRUN] DEALLOCATE DATABASE[S] FROM SERVER[S] 'name'[, ...]</code>
Description	Removes all databases from the given servers. For more information see Deallocate databases .
Required privilege	<code>GRANT SERVER MANAGEMENT</code> (see SERVER MANAGEMENT privileges)

Command	<code>DROP SERVER</code>
Syntax	<code>DROP SERVER 'name'</code>
Description	Removes a server not hosting any databases from the cluster. For more information see Drop server .
Required privilege	<code>GRANT SERVER MANAGEMENT</code> (see SERVER MANAGEMENT privileges)

Command	<code>SHOW SERVERS</code>
Syntax	<code>SHOW SERVER[S]</code> [YIELD { * field[, ...] } [ORDER BY field[, ...]] [SKIP n] [LIMIT n]] [WHERE expression] [RETURN field[, ...] [ORDER BY field[, ...]] [SKIP n] [LIMIT n]]
Description	Lists all servers visible to the cluster. For more information see Listing servers .
Required privilege	<code>GRANT SHOW SERVERS</code> (see SERVER MANAGEMENT privileges)

Listing servers

`SHOW SERVERS` displays all servers running in the cluster, including servers that have yet to be enabled as well as dropped servers.

The table of results shows information about the servers:

Column	Description	Default output	Full output
name	Name of the server.	✓	✓
serverId	Id of the server.		✓
address	Bolt address of the server (if enabled).	✓	✓
httpAddress	Http address of the server (if enabled).		✓
httpsAddress	Https address of the server (if enabled).		✓

Column	Description	Default output	Full output
state	Information of the state of the server: <code>free</code> , <code>enabled</code> , <code>deallocating</code> , or <code>dropped</code> .	✓	✓
health	The availability of the server: <code>available</code> or <code>unavailable</code> .	✓	✓
hosting	A list of databases currently hosted on the server.	✓	✓
requestedHosting	A list of databases that should be hosted on the server, decided by the allocator.		✓
tags	Tags are user provided strings that can be used while allocating databases.		✓
allowedDatabases	A list of databases allowed to be hosted on the server.		✓
deniedDatabases	A list of databases not allowed to be hosted on the server.		✓
modeConstraint	Constraint for the allocator to allocate only databases in this mode on the server.		✓
version	Neo4j version the server is running.		✓

A summary of all servers can be displayed using the command `SHOW SERVERS`.

Table 606. Result

name	address	state	health	hosting
"server1"	"localhost:20000"	"Enabled"	"Available"	["system", "neo4j"]
"server2"	"localhost:20007"	"Enabled"	"Available"	["system", "neo4j"]
"server3"	"localhost:20014"	"Enabled"	"Available"	["system", "neo4j"]
"0c030000-267b-49a8-801f-78bd0b5c6445"	"localhost:20021"	"Free"	"Available"	["system"]

Enabling servers

A server can be added to the cluster with the `ENABLE SERVER 'name'` command. The servers initial name is its id. The server must be in the `free` state to be added to the cluster. If the server is already `enabled` and the command is executed with the same options specified nothing is changed. In any other case trying to enable a server fails.

The possible options allowed when enabling a server are:

Option	Allowed values	Description
modeConstraint	PRIMARY, SECONDARY, NONE	Databases may only be hosted on the server in the mode specified by the constraint. None means there is no constraint and any mode is allowed.
allowedDatabases	list of database names	Only databases matching the specified names may be hosted on the server. This may not be specified in combination with deniedDatabases .
deniedDatabases	list of database names	Only databases not matching the specified names may be hosted on the server. This may not be specified in combination with allowedDatabases .



Composite databases are ignored by both **allowedDatabases** and **deniedDatabases**. The composite databases are available everywhere and hold no data on their own.

Modifying servers

The constraints on a server can be changed with `ALTER SERVER 'name' SET OPTIONS { option: value }`. Either the name or the id of the server can be used.

The possible options allowed when altering a server are:

Option	Allowed values	Description
modeConstraint	PRIMARY, SECONDARY, NONE	Databases may only be hosted on the server in the mode specified by the constraint. None means there is no constraint and any mode is allowed.
allowedDatabases	list of database names	Only databases matching the specified names may be hosted on the server. This may not be specified in combination with deniedDatabases .

Option	Allowed values	Description
deniedDatabases	list of database names	Only databases not matching the specified names may be hosted on the server. This may not be specified in combination with <code>allowedDatabases</code> .



Composite databases are ignored by both `allowedDatabases` and `deniedDatabases`. The composite databases are available everywhere and hold no data on their own.

Renaming servers

The name of a server can be altered with `RENAME SERVER 'name' TO 'newName'`. Either the id or current name of the server can be used to identify the server. The new name of the server must be unique.

Reallocate databases

After enabling a server, `REALLOCATE DATABASES` can be used to make the cluster re-balance databases across all servers that are part of the cluster. Using `DRYRUN REALLOCATE DATABASE` returns a view of how the databases would have been re-balanced if the command was executed without `DRYRUN`:

Table 607. Result

database	fromServerName	fromServerId	toServerName	toServerId	mode
"db1"	"server-1"	"00000000-94ff-4ede-87be-3d741b795480"	"server-4"	"00000002-25a9-4984-9ad2-dc39024c9238"	"primary"
"db3"	"server-1"	"00000000-94ff-4ede-87be-3d741b795480"	"server-5"	"00000003-0df7-4057-81fd-1cf43c9ef5f7"	"primary"



`DRYRUN` is introduced in Neo4j 5.2, and thus is not available in earlier minor releases of v5.

Deallocate databases

A server can be set to not host any databases with `DEALLOCATE DATABASES FROM SERVER 'name'`, in preparation for removing the server from the cluster. Either the id or name of the server can be used. All databases that the server is hosting are moved to other servers. The server changes state to `deallocating`. A deallocated server cannot readily be enabled again.

Multiple servers can be deallocated at the same time, `DEALLOCATE DATABASES FROM SERVER 'server-1', 'server-2'`. The command fails if there aren't enough servers available to move the databases to.

Using `DRYRUN DEALLOCATE DATABASES FROM 'server-1', 'server-2'` returns a view of how the databases would have been re-balanced if the command was executed without `DRYRUN`:

Table 608. Result

database	fromServerName	fromServerId	toServerName	toServerId	mode
"db1"	"server-1"	"00000001-8c04-4731-a2fd-7b0289c511ce"	"server-4"	"00000002-5b91-43c1-8b25-5289f674563e"	"primary"
"db1"	"server-2"	"00000000-7e53-427c-a987-24634c4745f3"	"server-5"	"00000003-0e98-44c8-9844-f0a4eb95b0d8"	"primary"

Drop server

When a server has been deallocated and is no longer hosting any databases it can be removed from the cluster with `DROP SERVER 'name'`. Either the id or name of the server can be used. As long as the server is running, it is listed when showing servers with the state `dropped`.

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 609. Result

command
"GRANT ACCESS ON HOME DATABASE TO `PUBLIC`"
"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 610. Result

command
"GRANT ACCESS ON DATABASE * TO `reader`"
"GRANT MATCH {*} ON GRAPH * NODE * TO `reader`"
"GRANT MATCH {*} ON GRAPH * RELATIONSHIP * TO `reader`"
"GRANT SHOW CONSTRAINT ON DATABASE * TO `reader`"
"GRANT SHOW INDEX ON DATABASE * TO `reader`"
Rows: 5

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
```

```
GRANT SHOW CONSTRAINT ON DATABASE * TO reader
```

```
GRANT SHOW INDEX ON DATABASE * 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 611. Result

command
"GRANT ACCESS ON DATABASE * TO `editor`"
"GRANT MATCH {*} ON GRAPH * NODE * TO `editor`"
"GRANT MATCH {*} ON GRAPH * RELATIONSHIP * TO `editor`"
"GRANT SHOW CONSTRAINT ON DATABASE * TO `editor`"
"GRANT SHOW INDEX ON DATABASE * TO `editor`"
"GRANT WRITE ON GRAPH * TO `editor`"
Rows: 6

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
```

```
GRANT SHOW CONSTRAINT ON DATABASE * TO editor
```

```
GRANT SHOW INDEX ON DATABASE * 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 612. 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 SHOW CONSTRAINT ON DATABASE * TO `publisher`"
"GRANT SHOW INDEX ON DATABASE * TO `publisher`"
"GRANT WRITE ON GRAPH * TO `publisher`"
Rows: 7

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
```

```
GRANT SHOW CONSTRAINT ON DATABASE * TO publisher
```

```
GRANT SHOW INDEX 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 613. 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 SHOW CONSTRAINT ON DATABASE * TO `architect`"
"GRANT SHOW INDEX ON DATABASE * TO `architect`"
"GRANT WRITE ON GRAPH * TO `architect`"
Rows: 9

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 SHOW CONSTRAINT ON DATABASE * TO architect
```

```
GRANT CONSTRAINT MANAGEMENT ON DATABASE * TO architect
```

```
GRANT SHOW INDEX ON DATABASE * TO architect
```

```
GRANT INDEX 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 614. 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 SHOW CONSTRAINT ON DATABASE * TO `admin`"
"GRANT SHOW INDEX 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: 13

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 effective privileges as the original built-in `admin` role.

Additional information about restoring the `admin` role can be found in the [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.



The syntax descriptions use [the style](#) from access control.

The `TRAVERSE` privilege

Users can be granted the right to find nodes and relationships using the `GRANT TRAVERSE` privilege.

```
GRANT [IMMUTABLE] 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 [IMMUTABLE] 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 [IMMUTABLE] 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 [IMMUTABLE] 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 [IMMUTABLE] 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 [IMMUTABLE] 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.



The syntax descriptions use [the style](#) from access control.

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 [IMMUTABLE] 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 [IMMUTABLE] 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 [IMMUTABLE] 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 [IMMUTABLE] 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 [IMMUTABLE] 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 [IMMUTABLE] 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 [IMMUTABLE] 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 [IMMUTABLE] 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 [IMMUTABLE] 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 [IMMUTABLE] 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 [IMMUTABLE] 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 [IMMUTABLE] 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 [IMMUTABLE] 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:

```
GRANT [IMMUTABLE] 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 [IMMUTABLE] 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.
- **mutability:**
 - **IMMUTABLE** - When used in conjunction with **GRANT** or **DENY**, specifies that a privilege cannot subsequently be removed unless auth is disabled. Contrastingly, when **IMMUTABLE** is specified in conjunction with a **REVOKE** command, it will act as a filter and only remove matching immutable privileges. See also [immutable privileges](#).
- **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.



The syntax descriptions use the style from access control.

Table 615. General grant ON DATABASE privilege syntax

Command	<code>GRANT ... ON ... TO ...</code>
Syntax	<code>GRANT [IMMUTABLE] database-privilege ON { HOME DATABASE DATABASE[S] { * name[, ...] } } TO role[, ...]</code>
Description	Grants a privilege to one or multiple roles.

Table 616. General deny ON DATABASE privilege syntax

Command	<code>DENY ... ON ... TO ...</code>
Syntax	<code>DENY [IMMUTABLE] database-privilege ON { HOME DATABASE DATABASE[S] { * name[, ...] } } TO role[, ...]</code>
Description	Denies a privilege to one or multiple roles.

Table 617. General revoke ON DATABASE privilege syntax

Command	<code>REVOKE GRANT ... ON ... FROM ...</code>
Syntax	<code>REVOKE [IMMUTABLE] GRANT database-privilege ON { HOME DATABASE DATABASE[S] { * name[, ...] } } FROM role[, ...]</code>
Description	Revoke a granted privilege from one or multiple roles.

Table 618. General revoke ON DATABASE privilege syntax

Command	<code>REVOKE DENY ... ON ... FROM ...</code>
Syntax	<code>REVOKE [IMMUTABLE] DENY database-privilege ON { HOME DATABASE DATABASE[S] { * name[, ...] } } FROM role[, ...]</code>
Description	Revokes a denied privilege from one or multiple roles.

Table 619. General revoke ON DATABASE privilege syntax

Command	<code>REVOKE ... ON ... FROM ...</code>
Syntax	<code>REVOKE [IMMUTABLE] database-privilege ON { HOME DATABASE DATABASE[S] { * name[, ...] } } FROM role[, ...]</code>
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 620. Database privilege syntax

Command	<code>GRANT ACCESS</code>
Syntax	<code>GRANT [IMMUTABLE] ACCESS ON { HOME DATABASE DATABASE[S] { * name[, ...] } } TO role[, ...]</code>
Description	<p>Grants the specified roles the privilege to access:</p> <ul style="list-style-type: none"> • The home database. • Specific database(s) or remote database alias(es). • All databases and remote database aliases.

Table 621. Database privilege syntax

Command	<code>GRANT { START STOP }</code>
Syntax	<code>GRANT [IMMUTABLE] { START STOP } ON { HOME DATABASE DATABASE[S] { * name[, ...] } } TO role[, ...]</code>

Description	Grants the specified roles the privilege to start or stop the home database, specific database(s), or all databases.
-------------	--

Table 622. Database privilege syntax

Command	<code>GRANT { CREATE DROP SHOW } INDEX</code>
Syntax	<pre>GRANT [IMMUTABLE] { 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 623. Database privilege syntax

Command	<code>GRANT INDEX</code>
Syntax	<pre>GRANT [IMMUTABLE] 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 624. Database privilege syntax

Command	<code>GRANT { CREATE DROP SHOW } CONSTRAINT</code>
Syntax	<pre>GRANT [IMMUTABLE] { CREATE DROP SHOW } CONSTRAINT[S] ON { HOME DATABASE DATABASE[S] { * name[, ...] } } TO role[, ...]</pre>
Description	Grants the specified roles the privilege to create, delete, or show constraints on the home database, specific database(s), or all databases.

Table 625. Database privilege syntax

Command	<code>GRANT CONSTRAINT</code>
Syntax	<pre>GRANT [IMMUTABLE] CONSTRAINT[S] [MANAGEMENT] ON { HOME DATABASE DATABASE[S] { * name[, ...] } } TO role[, ...]</pre>
Description	Grants the specified roles the privilege to manage constraints on the home database, specific database(s), or all databases.

Table 626. Database privilege syntax

Command	<code>GRANT CREATE NEW LABEL</code>
---------	-------------------------------------

Syntax	<pre>GRANT [IMMUTABLE] CREATE NEW [NODE] LABEL[S] ON { HOME DATABASE DATABASE[S] { * name[, ...] } } TO role[, ...]</pre>
Description	Grants the specified roles the privilege to create new node labels in the home database, specific database(s), or all databases.

Table 627. Database privilege syntax

Command	<code>GRANT CREATE NEW TYPE</code>
Syntax	<pre>GRANT [IMMUTABLE] CREATE NEW [RELATIONSHIP] TYPE[S] ON { HOME DATABASE DATABASE[S] { * name[, ...] } } TO role[, ...]</pre>
Description	Grants the specified roles the privilege to create new relationship types in the home database, specific database(s), or all databases.

Table 628. Database privilege syntax

Command	<code>GRANT CREATE NEW NAME</code>
Syntax	<pre>GRANT [IMMUTABLE] CREATE NEW [PROPERTY] NAME[S] ON { HOME DATABASE DATABASE[S] { * name[, ...] } } TO role[, ...]</pre>
Description	Grants the specified roles the privilege to create new property names in the home database, specific database(s), or all databases.

Table 629. Database privilege syntax

Command	<code>GRANT NAME</code>
Syntax	<pre>GRANT [IMMUTABLE] 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 630. Database privilege syntax

Command	<code>GRANT ALL</code>
Syntax	<pre>GRANT [IMMUTABLE] 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 631. Database privilege syntax

Command	<code>GRANT { SHOW TERMINATE } TRANSACTION</code>
Syntax	<code>GRANT [IMMUTABLE] { SHOW TERMINATE } TRANSACTION[S] [({ * user[, ...] })] ON { HOME DATABASE DATABASE[S] { * name[, ...] } } TO role[, ...]</code>
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 632. Database privilege syntax

Command	<code>GRANT TRANSACTION</code>
Syntax	<code>GRANT [IMMUTABLE] TRANSACTION [MANAGEMENT] [({ * user[, ...] })] ON { HOME DATABASE DATABASE[S] { * name[, ...] } } TO role[, ...]</code>
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.

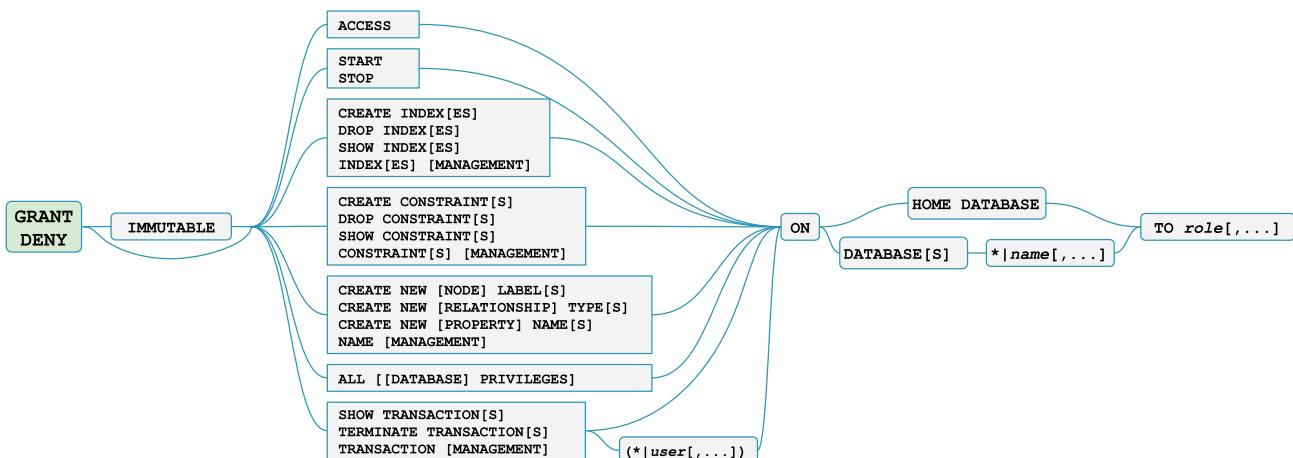


Figure 5. Syntax of GRANT and DENY Database Privileges

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 [IMMUTABLE] 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 [IMMUTABLE] 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 633. Result

command
"DENY ACCESS ON DATABASE remote-db TO `regularUsers`"
"GRANT ACCESS ON DATABASE neo4j TO `regularUsers`"
Rows: 2

The database **START/STOP** privileges

The **START** privilege can be used to enable the ability to start a database:

```
GRANT [IMMUTABLE] 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 [IMMUTABLE] 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 [IMMUTABLE] 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 [IMMUTABLE] 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 634. 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 635. Index management privilege syntax

Command	GRANT { CREATE DROP SHOW } INDEX
---------	--------------------------------------

Syntax	<pre>GRANT [IMMUTABLE] { 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 636. Index management privilege syntax

Command	GRANT INDEX
Syntax	<pre>GRANT [IMMUTABLE] 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 **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 637. Constraint management privilege syntax

Command	GRANT { CREATE DROP SHOW } CONSTRAINT
Syntax	<pre>GRANT [IMMUTABLE] { 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 638. Constraint management privilege syntax

Command	GRANT CONSTRAINT
Syntax	<pre>GRANT [IMMUTABLE] 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 `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 639. Node label management privileges syntax

Command	<code>GRANT CREATE NEW LABEL</code>
Syntax	<pre>GRANT [IMMUTABLE] 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 640. Relationship type management privileges syntax

Command	<code>GRANT CREATE NEW TYPE</code>
Syntax	<pre>GRANT [IMMUTABLE] 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 641. Property name management privileges syntax

Command	<code>GRANT CREATE NEW NAME</code>
Syntax	<pre>GRANT [IMMUTABLE] 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 642. Node label, relationship type, and property name privileges management syntax

Command	<code>GRANT NAME</code>
Syntax	<pre>GRANT [IMMUTABLE] 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 [IMMUTABLE] 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 643. 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 644. Database privilege syntax

Command	<code>GRANT SHOW TRANSACTION</code>
Syntax	<pre>GRANT [IMMUTABLE] 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 645. Database privilege syntax

Command	<code>GRANT TERMINATE TRANSACTION</code>
Syntax	<pre>GRANT [IMMUTABLE] 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 646. Database privilege syntax

Command	<code>GRANT TRANSACTION</code>
Syntax	<pre>GRANT [IMMUTABLE] 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 [Neo4j databases and graphs](#).

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 647. 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 648. 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.



The syntax descriptions use the style from access control.

Table 649. Role management privileges command syntax

Command	Description
GRANT [IMMUTABLE] CREATE ROLE ON DBMS TO role[, ...]	Enables the specified roles to create new roles.
GRANT [IMMUTABLE] RENAME ROLE ON DBMS TO role[, ...]	Enables the specified roles to change the name of roles.
GRANT [IMMUTABLE] DROP ROLE ON DBMS TO role[, ...]	Enables the specified roles to delete roles.
GRANT [IMMUTABLE] ASSIGN ROLE ON DBMS TO role[, ...]	Enables the specified roles to assign roles to users.
GRANT [IMMUTABLE] REMOVE ROLE ON DBMS TO role[, ...]	Enables the specified roles to remove roles from users.
GRANT [IMMUTABLE] SHOW ROLE ON DBMS TO role[, ...]	Enables the specified roles to list roles.
GRANT [IMMUTABLE] ROLE MANAGEMENT ON DBMS TO role[, ...]	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 650. 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 651. 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 652. 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 653. 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 654. 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 655. 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 656. 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.



The syntax descriptions use the style from access control.

Table 657. User management privileges command syntax

Command	Description
<pre>GRANT [IMMUTABLE] CREATE USER ON DBMS TO role[, ...]</pre>	Enables the specified roles to create new users.
<pre>GRANT [IMMUTABLE] RENAME USER ON DBMS TO role[, ...]</pre>	Enables the specified roles to change the name of users.
<pre>GRANT [IMMUTABLE] ALTER USER ON DBMS TO role[, ...]</pre>	Enables the specified roles to modify users.
<pre>GRANT [IMMUTABLE] 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 [IMMUTABLE] SET USER HOME DATABASE ON DBMS TO role[, ...]</pre>	Enables the specified roles to modify users' home database.

Command	Description
GRANT [IMMUTABLE] SET USER STATUS ON DBMS TO role[, ...]	Enables the specified roles to modify the account status of users.
GRANT [IMMUTABLE] DROP USER ON DBMS TO role[, ...]	Enables the specified roles to delete users.
GRANT [IMMUTABLE] SHOW USER ON DBMS TO role[, ...]	Enables the specified roles to list users.
GRANT [IMMUTABLE] USER MANAGEMENT ON DBMS TO role[, ...]	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 658. Result

command
<code>"GRANT CREATE USER ON DBMS TO `userAdder`"</code>
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 659. Result

command
"GRANT RENAME USER ON DBMS TO `userModifier`"
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 660. 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 661. 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 662. 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 663. 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 664. 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 665. 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 666. 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.



The syntax descriptions use [the style](#) from access control.

Table 667. Impersonation privileges command syntax

Command	Description
<pre>GRANT [IMMUTABLE] IMPERSONATE [(*)] ON DBMS TO role[, ...]</pre>	Enables the specified roles to impersonate any user.
<pre>GRANT [IMMUTABLE] 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 668. 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.



The syntax descriptions use [the style](#) from access control.

Table 669. Database management privileges command syntax

Command	Description
<pre>GRANT [IMMUTABLE] CREATE DATABASE ON DBMS TO role[, ...]</pre>	Enables the specified roles to create new standard databases and aliases.
<pre>GRANT [IMMUTABLE] DROP DATABASE ON DBMS TO role[, ...]</pre>	Enables the specified roles to delete standard databases and aliases.
<pre>GRANT [IMMUTABLE] ALTER DATABASE ON DBMS TO role[, ...]</pre>	Enables the specified roles to modify standard databases and aliases.

Command	Description
GRANT [IMMUTABLE] SET DATABASE ACCESS ON DBMS TO role[, ...]	Enables the specified roles to modify access to standard databases.
GRANT CREATE COMPOSITE DATABASE ON DBMS TO role[, ...]	Enables the specified roles to create new composite databases.
GRANT DROP COMPOSITE DATABASE ON DBMS TO role[, ...]	Enables the specified roles to delete composite databases.
GRANT COMPOSITE DATABASE MANAGEMENT ON DBMS TO role[, ...]	Enables the specified roles to create and delete composite databases.
GRANT [IMMUTABLE] DATABASE MANAGEMENT ON DBMS TO role[, ...]	Enables the specified roles to create, delete, and modify databases and aliases.

The ability to create standard 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 standard databases and aliases. List all privileges for the role `databaseAdder` as commands by using the following query:

```
SHOW ROLE databaseAdder PRIVILEGES AS COMMANDS
```

Table 670. Result

command
<code>"GRANT CREATE DATABASE ON DBMS TO `databaseAdder`"</code>
Rows: 1

The ability to create composite databases can be granted via the `CREATE COMPOSITE DATABASE` privilege. See an example:

```
GRANT CREATE COMPOSITE DATABASE ON DBMS TO compositeDatabaseAdder
```

The resulting role has privileges that only allow creating composite databases. List all privileges for the role

`compositeDatabaseAdder` as commands by using the following query:

```
SHOW ROLE compositeDatabaseAdder PRIVILEGES AS COMMANDS
```

Table 671. Result

command
"GRANT CREATE COMPOSITE DATABASE ON DBMS TO `compositeDatabaseAdder`"
Rows: 1

The ability to delete standard 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 standard databases and aliases. List all privileges for the role `databaseDropper` as commands by using the following query:

```
SHOW ROLE databaseDropper PRIVILEGES AS COMMANDS
```

Table 672. Result

command
"GRANT DROP DATABASE ON DBMS TO `databaseDropper`"
Rows: 1

The ability to delete composite databases can be granted via the `DROP COMPOSITE DATABASE` privilege. See an example:

```
GRANT DROP COMPOSITE DATABASE ON DBMS TO compositeDatabaseDropper
```

The resulting role has privileges that only allow deleting composite databases. List all privileges for the role `compositeDatabaseDropper` as commands by using the following query:

```
SHOW ROLE compositeDatabaseDropper PRIVILEGES AS COMMANDS
```

Table 673. Result

command
"GRANT DROP COMPOSITE DATABASE ON DBMS TO `compositeDatabaseDropper`"
Rows: 1

The ability to modify standard 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 standard databases and aliases. List all privileges for the role `databaseModifier` as commands by using the following query:

```
SHOW ROLE databaseModifier PRIVILEGES AS COMMANDS
```

Table 674. Result

command
"GRANT ALTER DATABASE ON DBMS TO `databaseModifier`"

Rows: 1

The ability to modify access to standard 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 to standard databases. List all privileges for the role `accessModifier` as commands by using the following query:

```
SHOW ROLE accessModifier PRIVILEGES AS COMMANDS
```

Table 675. Result

command
"GRANT SET DATABASE ACCESS ON DBMS TO `accessModifier`"

Rows: 1

The ability to create and delete composite databases can be granted via the `COMPOSITE DATABASE MANAGEMENT` privilege. See an example:

```
GRANT COMPOSITE DATABASE MANAGEMENT ON DBMS TO compositeDatabaseManager
```

The resulting role has all privileges to manage composite databases. List all privileges for the role `compositeDatabaseManager` as commands by using the following query:

```
SHOW ROLE compositeDatabaseManager PRIVILEGES AS COMMANDS
```

Table 676. Result

command

```
"GRANT COMPOSITE DATABASE MANAGEMENT ON DBMS TO `compositeDatabaseManager`"
```

Rows: 1

The ability 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 standard and composite databases as well as aliases. List all privileges for the role **databaseManager** as commands by using the following query:

```
SHOW ROLE databaseManager PRIVILEGES AS COMMANDS
```

Table 677. Result

command

```
"GRANT DATABASE MANAGEMENT ON DBMS TO `databaseManager`"
```

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](#).



The syntax descriptions use [the style](#) from access control.

Table 678. Alias management privileges command syntax

Command	Description
<pre>GRANT [IMMUTABLE] CREATE ALIAS ON DBMS TO role[, ...]</pre>	Enables the specified roles to create new aliases.
<pre>GRANT [IMMUTABLE] DROP ALIAS ON DBMS TO role[, ...]</pre>	Enables the specified roles to delete aliases.
<pre>GRANT [IMMUTABLE] ALTER ALIAS ON DBMS TO role[, ...]</pre>	Enables the specified roles to modify aliases.

Command	Description
GRANT [IMMUTABLE] SHOW ALIAS ON DBMS TO role[, ...]	Enables the specified roles to list aliases.
GRANT [IMMUTABLE] ALIAS MANAGEMENT ON DBMS TO role[, ...]	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 679. 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 680. 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 681. 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 682. 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 683. Result

command
"GRANT ALIAS MANAGEMENT ON DBMS TO `aliasManager`"
Rows: 1

The DBMS SERVER MANAGEMENT privileges

The DBMS privileges for server management can be assigned using Cypher administrative commands. They can be granted, denied, and revoked like other privileges.



The syntax descriptions use [the style](#) from access control.

Table 684. Server management privileges command syntax

Command	Description
<pre>GRANT SERVER MANAGEMENT ON DBMS TO role[, ...]</pre>	Enables the specified roles to show, enable, rename, alter, reallocate, deallocate, and drop servers.
<pre>GRANT SHOW SERVERS ON DBMS TO role[, ...]</pre>	Enables the specified roles to show servers.

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.



The syntax descriptions use [the style](#) from access control.

Table 685. Privilege management privileges command syntax

Command	Description
<pre>GRANT [IMMUTABLE] SHOW PRIVILEGE ON DBMS TO role[, ...]</pre>	Enables the specified roles to list privileges.
<pre>GRANT [IMMUTABLE] ASSIGN PRIVILEGE ON DBMS TO role[, ...]</pre>	Enables the specified roles to assign privileges using the GRANT and DENY commands.
<pre>GRANT [IMMUTABLE] REMOVE PRIVILEGE ON DBMS TO role[, ...]</pre>	Enables the specified roles to remove privileges using the REVOKE command.
<pre>GRANT [IMMUTABLE] 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 686. Result

command
<code>"GRANT SHOW PRIVILEGE ON DBMS TO `privilegeShower`"</code>
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 687. Result

command
<code>"GRANT ASSIGN PRIVILEGE ON DBMS TO `privilegeAssigner`"</code>
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 688. Result

command
<code>"GRANT REMOVE PRIVILEGE ON DBMS TO `privilegeRemover`"</code>
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 689. Result

command
<code>"GRANT PRIVILEGE MANAGEMENT ON DBMS TO `privilegeManager`"</code>
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.



The syntax descriptions use [the style](#) from access control.

Table 690. Execute privileges command syntax

Command	Description
<pre>GRANT [IMMUTABLE] EXECUTE PROCEDURE[S] name-globbing[, ...] ON DBMS TO role[, ...]</pre>	Enables the specified roles to execute the given procedures.
<pre>GRANT [IMMUTABLE] 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 [IMMUTABLE] EXECUTE ADMINISTRATOR PROCEDURES ON DBMS TO role[, ...]</pre>	Enables the specified roles to execute procedures annotated with <code>@Admin</code> . The procedures are executed with elevated privileges.
<pre>GRANT [IMMUTABLE] 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 [IMMUTABLE] 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 691. 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 692. 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 PROCEDURE * ON DBMS TO boostedProcedureExecutor  
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 693. Result

command
"GRANT EXECUTE PROCEDURE * ON DBMS TO `boostedProcedureExecutor`"
"GRANT EXECUTE BOOSTED PROCEDURE db.labels ON DBMS TO `boostedProcedureExecutor`"
"GRANT EXECUTE BOOSTED PROCEDURE db.relationshipTypes ON DBMS TO `boostedProcedureExecutor`"
Rows: 3

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 407. 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 694. Result

command
"DENY EXECUTE BOOSTED PROCEDURE db.labels ON DBMS TO `deniedBoostedProcedureExecutor1`"
"GRANT EXECUTE PROCEDURE * ON DBMS TO `deniedBoostedProcedureExecutor1`"
Rows: 2

Example 408. 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 695. Result

command
"DENY EXECUTE PROCEDURE db.labels ON DBMS TO `deniedBoostedProcedureExecutor2`"
"GRANT EXECUTE BOOSTED PROCEDURE * ON DBMS TO `deniedBoostedProcedureExecutor2`"
Rows: 2

Example 409. 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 696. Result

command
"DENY EXECUTE BOOSTED PROCEDURE db.labels ON DBMS TO `deniedBoostedProcedureExecutor3`"
"GRANT EXECUTE BOOSTED PROCEDURE * ON DBMS TO `deniedBoostedProcedureExecutor3`"
Rows: 2

Example 410. 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 697. 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 411. 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.

For comparison, when granted:

- `EXECUTE PROCEDURE myProc`: the `myProc` procedure returns the result A and B.
- `EXECUTE BOOSTED PROCEDURE myProc`: execution of the `myProc` procedure is not allowed.
- `EXECUTE PROCEDURE myProc` and `EXECUTE BOOSTED PROCEDURE myProc`: the `myProc` procedure returns the result A, B, and C.

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 698. 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 412. 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 699. 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 413. 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 700. 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 414. Execute boosted user-defined function

The following query shows an example of how to grant the `EXECUTE BOOSTED USER DEFINED FUNCTION` privilege:

```
GRANT EXECUTE USER DEFINED FUNCTION * ON DBMS TO boostedFunctionExecutor  
GRANT EXECUTE BOOSTED USER DEFINED FUNCTION apoc.any.properties ON DBMS TO boostedFunctionExecutor
```

Or in short form:

```
GRANT EXECUTE FUNCTION * ON DBMS TO boostedFunctionExecutor  
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 701. Result

command
"GRANT EXECUTE FUNCTION * ON DBMS TO `boostedFunctionExecutor`"
"GRANT EXECUTE BOOSTED FUNCTION apoc.any.properties ON DBMS TO `boostedFunctionExecutor`"
Rows: 2

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 and aliases.
- Enable, alter, rename, reallocate, deallocate, and drop servers
- Show, assign, and remove privileges.
- Execute all procedures with elevated privileges.
- Execute all user defined functions with elevated privileges.



The syntax descriptions use the style from access control.

```
GRANT [IMMUTABLE] 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 702. 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 5 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 free role	User with restricted role
<p>The database can access the count store and retrieve the total number of nodes with the label <code>:Person</code>.</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 <code>:Person</code> are counted. Due to this, each node with the <code>:Person</code> label needs to be accessed and examined to make sure that they do not have a deny-listed label, such as <code>:Customer</code>.</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>

Immutable privileges

Unlike regular privileges, having [privilege management](#) privileges is not sufficient to enable immutable privileges to be added or removed. They can only be administered when auth is disabled —that is, when the configuration setting `dbms.security.auth_enabled` is set to `false`.

When to use immutable privileges

Immutable privileges are useful for restricting the actions of users who themselves are able to [administer privileges](#).

For example, you may want to prevent all users from performing Database Management, even the `admin` user (who are themselves able to add or remove privileges). To do so, you could run:

```
DENY DATABASE MANAGEMENT ON DBMS TO PUBLIC
```

However, this would not be adequate. In case the `admin` user subsequently runs this:

```
REVOKE DENY DATABASE MANAGEMENT ON DBMS TO PUBLIC
```

They would effectively regain Database Management privileges. Instead, run the following query to prevent this scenario:

```
DENY IMMUTABLE DATABASE MANAGEMENT ON DBMS TO PUBLIC
```

How to administer immutable privileges

Immutable privileges can only be administered when auth is disabled — that is, when the configuration setting `dbms.security.auth_enabled` is set to `false`, for example. Under these conditions, immutable privileges can be added and removed in a similar manner to regular privileges, using the `IMMUTABLE` keyword.

See the [Immutable privileges tutorial](#) for examples of how to administer immutable privileges.

See [Managing Privileges](#) for more detail on syntax.

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

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 runtime

The Cypher runtime runs queries and returns records based on an execution plan. Depending on the Neo4j edition, there are two different runtimes available:

Slotted

In the slotted 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 comprises nested iterators, which stream records from the top iterator, which pulls from the next iterator, and so on. Each variable in the query gets a dedicated "slot" or offset, which the runtime uses for accessing (e.g., `slotRow[0]`).

The slotted runtime covers all operators and queries.

In Neo4j 5, it is the default for Community Edition.

Pipelined

In the pipelined runtime, the operators are grouped into pipelines in the execution plan to generate new

combinations and orders of execution, which are optimized for performance and memory usage.

The pipelined runtime covers most operators and queries. If the `pipelined` runtime does not support a query, the planner falls back to the `slotted` runtime.

It is the default for Enterprise Edition.

Option	Description	Default
<code>runtimetype=slotted</code>	Forces the Cypher query planner to use the slotted runtime.	Default for Community Edition.
<code>runtimetype=pipelined</code>	Forces the Cypher query planner to use the pipelined runtime.	Default for Enterprise 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.	



Using this option can significantly increase the planning time of the query.

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>	<p>Use a greedy approach when combining sub-plans.</p> <p> Using this option can significantly reduce the planning time of the query.</p>	
<code>connectComponentsPlanner=idp</code>	<p>Use the cost based IDP search algorithm when combining sub-plans.</p> <p> Using this option can significantly increase the planning time of the query but usually finds better plans.</p>	✓

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 interpreted expression engine.	

Query option	Description	Default
<code>expressionEngine=compiled</code>	Always compile expressions and use the compiled 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 compiled 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> .	

Query option	Description	Default
<code>interpretedPipesFallback=whitelisted_plans_only</code>	<p>Parts of the execution plan can be executed on another runtime. Only certain operators are allowed to execute on another runtime.</p> <p>Cannot be used together with <code>runtime=interpreted</code> or <code>runtime=slotted</code>.</p>	
<code>interpretedPipesFallback=all</code>	<p>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.</p> <p>Cannot be used together with <code>runtime=interpreted</code> or <code>runtime=slotted</code>.</p> <p> This setting is experimental, and using it in a production environment is discouraged.</p>	

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 `server.db.query_cache_size` was exceeded.
- When the time has past the `dbms.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.	<input checked="" type="checkbox"/>

Option	Description	Default
<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 depends 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.

For instructions on how to create and maintain indexes, refer to [Indexes for search performance](#).

Index types and predicate compatibility

Generally, an index solves some combination of a label/relationship type predicate and property predicates at the same time. There are different types of indexes available in Neo4j and these are compatible with different property predicates.

Indexes are most often used for `MATCH` and `OPTIONAL MATCH` clauses that combine a label/relationship type predicate with a property predicate. Therefore, it is important to know what kind of predicates can be solved by the different indexes.

The different index types used for search performance are:

- `LOOKUP`
- `RANGE`
- `POINT`
- `TEXT`
- `BTREE` Deprecated



The `RANGE` and `TEXT` indexes can only perform limited matching on strings - exact, prefix, substring, or suffix matches. A `FULLTEXT` index will instead tokenize the indexed string values, so it can match terms anywhere within the strings. See [Full-text search index](#).

LOOKUP indexes

`LOOKUP` indexes are present by default and solve only node label and relationship type predicates:

Predicate	Syntax (example)
Node label predicate.	[source, syntax, role="noheader"] ---- MATCH (n:Label) ----
Node label predicate.	[source, syntax, role="noheader"] ---- MATCH (n) WHERE n:Label ----
Relationship type predicate.	[source, syntax, role="noheader"] ---- MATCH ()-[r:REL]>()
Relationship type predicate.	[source, syntax, role="noheader"] ---- MATCH ()-[r]>()
	WHERE r:REL ----



LOOKUP indexes are the most important index type in the database because they improve the performance of the Cypher queries and the population of other indexes. Dropping these indexes may lead to severe performance degradation. Therefore, carefully consider the consequences before doing so.

RANGE indexes

In combination with node label and relationship type predicates, **RANGE** indexes support most types of predicates:

Predicate	Syntax
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

POINT indexes

In combination with node label and relationship type predicates, **POINT** indexes only solve predicates operating on points. Therefore, **POINT** indexes are only used when it is known that the predicate evaluates to **null** for all non-point values.

POINT indexes only support point type predicates:

Predicate	Syntax
Property point value.	<code>n.prop = point({x: value, y: value})</code>
Within bounding box.	<code>point.withinBBox(n.prop, lowerLeftCorner, upperRightCorner)</code>
Distance.	<code>point.distance(n.prop, center) <= distance</code>

TEXT indexes

In combination with node label and relationship type predicates, **TEXT** indexes only solve 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"]`

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 = 'example_string'</code>
List membership check.	<code>n.prop IN ['abc', 'example_string', 'neo4j']</code>
Prefix search.	<code>STARTS WITH</code>
Suffix search.	<code>ENDS WITH</code>

Predicate	Syntax
Substring search.	CONTAINS

In some cases, the system cannot determine whether an expression is of type string.

For example when the compared value is a parameter:

```
MATCH (n:Label) WHERE n.prop = $param
```

Such queries can be modified to provide this information. Depending on how values that are not of type string should be treated, there are two options:

- If rows in which the expression is not of type string should be discarded, then adding `WHERE <expression> STARTS WITH ''` is the right option: `MATCH (n:Label) WHERE $param STARTS WITH '' AND n.prop = $param`
- If expressions which are not of type string should be converted to string, then wrapping these in `toString(<expression>)` is the right choice: `MATCH (n:Label) WHERE n.prop = toString($param)`

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 used over `RANGE` and `POINT` indexes for `CONTAINS` and `ENDS WITH`.
- `POINT` indexes are used over `RANGE` and `TEXT` indexes for distance and within a bounding box.
- `RANGE` indexes are preferred over `TEXT` and `POINT` indexes in all other cases.

`LOOKUP` indexes are not defined in this order since they never solve the same set of predicates as the other indexes.

Examples:

- [Node label LOOKUP index example](#)
- [Relationship type LOOKUP index example](#)
- [Node RANGE index example](#)
- [Relationship RANGE index example](#)
- [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 bounding box searches (single-property index)

Node label LOOKUP index example

In the example below, a node LOOKUP index is available.

Query

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

Query Plan

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

+-----+-----+-----+-----+
| Operator      | Details      | Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache
Hits/Misses | Time (ms) | Pipeline      |
+-----+-----+-----+-----+
| +ProduceResults | person      |           42 | 42 | 0 |          | |
| |               |             |           42 | 42 | 43 | 120 |
| |               |             |           42 | 42 | 43 | 120 |
| +NodeByLabelScan | person:Person |           42 | 42 | 43 | 120 |
2/1 | 0.565 | Fused in Pipeline 0 |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
Total database accesses: 43, total allocated memory: 184
```

Relationship type LOOKUP index example

In the example below, a relationship **LOOKUP** index is available.

Query

```
MATCH ()-[r:KNOWS]->()
RETURN r
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator (Bytes)	Details	Estimated Rows	Rows	DB Hits	Memory
Page Cache Hits/Misses	Time (ms)				
+ProduceResults	r	22	22	0	
+DirectedRelationshipTypeScan	(anon_0)-[r:KNOWS]->(anon_1)	22	22	23	
120	3/1 0.915 Fused in Pipeline 0				

Total database accesses: 23, total allocated memory: 184

Node RANGE index example

In the example below, a **Person(firstname)** node **RANGE** index is available.

Query

```
MATCH (person:Person {firstname: 'Andy'})
RETURN person
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows				
Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline	
+ProduceResults	person	1				
	0					1
+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+
+NodeIndexSeek	RANGE INDEX person:Person(firstname) WHERE firstname = \$autostring_0	1				
	2	120		2/1	0.635	Fused in Pipeline 0
+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+

Total database accesses: 2, total allocated memory: 184

Relationship RANGE index example

In this example, a KNOWS(since) relationship RANGE index is available.

Query

```
MATCH (person)-[relationship:KNOWS {since: 1992}]->(friend)  
RETURN person, friend
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline	
+ProduceResults	person, friend	1	1	0					
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
+DirectedRelationshipIndexSeek	RANGE INDEX (person)-[relationship:KNOWS(since)]->(friend) WHERE since = \$autoint_0	1	1	2	120		2/1	0.413	
									Fused in Pipeline 0
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+

Total database accesses: 2, total allocated memory: 184

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

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows
Rows	DB Hits Memory (Bytes) Page Cache Hits/Misses Time (ms) Pipeline	
+ProduceResults	person	1
1	0	
+NodeIndexSeek	TEXT INDEX person:Person(surname) WHERE surname = \$autostring_0 1 1	
1	2 120 2/0 0.446 Fused in Pipeline 0	

Total database accesses: 2, total allocated memory: 184

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

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details						Pipeline
	Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	
+ProduceResults	1	1	0	person, friend			
+DirectedRelationshipIndexSeek	1	1	2	TEXT INDEX (person)-[relationship:KNOWS(metIn)]->(friend) WHERE metIn = \$autostring_0	120	2/0	0.691
Fused in Pipeline 0							

Total database accesses: 2, total allocated memory: 184

Multiple available index types

In the example below, both a `Person(middlename)` node TEXT index and a `Person(middlename)` node RANGE index are available. The RANGE node index is chosen.

Query

```
MATCH (person:Person {middlename: 'Ron'})
RETURN person
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated					
Rows	Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline	
+ProduceResults	person						
1	1	0					
+NodeIndexSeek	RANGE INDEX person:Person(middlename) WHERE middlename = \$autostring_0						
1	1	2	120	2/1	0.423	Fused in Pipeline	0

Total database accesses: 2, total allocated memory: 184

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

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows				
Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline	
+ProduceResults	person	1				
	1 0					
+NodeIndexSeek	RANGE INDEX person:Person(firstname) WHERE firstname = \$autostring_0	1				
	1 2 120		2/1 0.292 Fused in Pipeline 0			

Total database accesses: 2, total allocated memory: 184

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](#).

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.

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

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details						Pipeline
	Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	
+ProduceResults	1	1	0	friend, person			
+DirectedRelationshipIndexSeekByRange	1	1	2	RANGE INDEX (person)-[r:KNOWS(since)]->(friend) WHERE since < \$autoInt_0	120	2/1	0.943 Fused in Pipeline 0

Total database accesses: 2, total allocated memory: 184

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

Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

Operator Details						
Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline
+ProduceResults	2	2	0	`r.since`		
+Projection	2	2	0	cache[r.since] AS `r.since`		
+Filter	2	2	0	cache[r.lastMet] > \$autoint_1		
+UndirectedRelationshipIndexSeek	RANGE INDEX (anon_0)-[r:KNOWS(since, lastMet)]-(anon_1) WHERE since < \$autoint_0 AND lastMet IS NOT	2	2	2 120 1/1	0.525 Fused in Pipeline 0	
				NULL, cache[r.since], cache[r.lastMet]		
Total database accesses: 2, total allocated memory: 184						

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

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

+-----+
+-----+
+-----+-----+-----+-----+
| Operator      | Details
| Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline
|-----|
+-----+
+-----+-----+-----+-----+
| +ProduceResults | person
|     1 |   1 |     0 |
|-----|
|-----+
+-----+-----+-----+
| +NodeIndexSeekByRange | RANGE INDEX person:Person(highScore) WHERE highScore > $autoint_0 AND highScore
< $autoint_1 |           1 |   1 |     2 |       120 |        2/1 |    0.286 |
Fused in Pipeline 0 |
+-----+
+-----+-----+-----+-----+
+-----+-----+-----+-----+
Total database accesses: 2, total allocated memory: 184
```

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

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

+-----+
+-----+
+-----+-----+-----+-----+
| Operator      | Details
| Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline
|-----|
+-----+
+-----+-----+-----+-----+
| +ProduceResults | person
|   1 | 1 | 0 |           |           |           |
|   |   |   |           |           |           |
|   |   |
+-----+
+-----+-----+-----+-----+
| +NodeIndexSeek | RANGE INDEX person:Person(highScore, name) WHERE highScore > $autoint_0 AND highScore
< $autoint_1 A |           1 | 1 | 2 | 120 | 2/1 | 4.498 |
Fused in Pipeline 0 |
|           ND name IS NOT NULL
|           |   |   |           |           |           |
|           |   |   |
+-----+
+-----+-----+-----+-----+
+-----+
Total database accesses: 2, total allocated memory: 184
```

List membership check using **IN** (single-property index)

The **IN** predicate on **r.since** in the following query will use the single-property index **KNOWS(lastMetIn)** if it exists.

Query

```
MATCH (person)-[r:KNOWS]->(friend)
WHERE r.lastMetIn IN ['Malmo', 'Stockholm']
RETURN person, friend
```

Query Plan

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

+-----+
+-----+-----+-----+-----+
| Operator | Details
| Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline
+-----+
+-----+-----+-----+-----+
| +ProduceResults | person, friend
|   1 | 1 | 0 |           |
|   |   |   |           |
|   |   |   |
+-----+
+-----+-----+-----+-----+
| +DirectedRelationshipIndexSeek | RANGE INDEX (person)-[r:KNOWS(lastMetIn)]->(friend) WHERE lastMetIn IN
$autolist_0 |           1 | 1 | 3 | 120 |           3/1 | 0.614 |
Fused in Pipeline 0 |
+-----+
+-----+-----+-----+-----+
Total database accesses: 3, total allocated memory: 184
```

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

Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

Operator Details						
Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline
+ProduceResults	1	1	0	person, friend		
+DirectedRelationshipIndexSeek	1.864	RANGE INDEX (person)-[r:KNOWS(since, lastMet)]->(friend) WHERE since IN \$autolist_0 AND lastMet IN \$1.864 Fused in Pipeline 0	1	1	5 120 5/1	
				autolist_1		

Total database accesses: 5, total allocated memory: 184

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

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

+-----+
+-----+-----+-----+-----+
| Operator      | Details
| Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline
| |
+-----+
+-----+-----+-----+-----+
| +ProduceResults    | person
| 2 |   1 |     0 |           |           |           |           |
| |           +-----+
+-----+-----+-----+-----+
| +NodeIndexSeekByRange | RANGE INDEX person:Person(firstname) WHERE firstname STARTS WITH $autostring_0 |
| 2 |   1 |     2 |           120 |           3/0 |     0.387 | Fused in Pipeline 0 |
+-----+
+-----+-----+-----+-----+
+-----+
Total database accesses: 2, total allocated memory: 184
```

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

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

+-----+
+-----+
+-----+-----+-----+-----+
| Operator      | Details
| Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline
|-----|
+-----+
+-----+-----+-----+-----+
| +ProduceResults | person
|   1 | 1 | 0 |           |           |           |
|   |   |   |           |           |           |
|   |   |
+-----+
+-----+-----+-----+-----+
| +NodeIndexSeek | RANGE INDEX person:Person(firstname, surname) WHERE firstname STARTS WITH
$autostring_0 AND surname |           1 | 1 | 2 | 120 |           3/0 |
0.534 | Fused in Pipeline 0 |
|           | IS NOT NULL
|           |           |           |           |           |
|           |           |           |           |           |
+-----+
+-----+-----+-----+-----+
+-----+
Total database accesses: 2, total allocated memory: 184
```

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. Text indexes are optimized for `CONTAINS` and `ENDS WITH` and they are the only indexes that can solve those predicates.

Query

```
MATCH (person)-[r:KNOWS]->(friend)
WHERE r.metIn ENDS WITH 'mo'
RETURN person, friend
```

Text indexes only index String values and therefore do not find other values.

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

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

+-----+
+-----+
+-----+-----+-----+-----+
| Operator | Details
| Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline
|          |
+-----+
+-----+-----+-----+-----+
+-----+
| +ProduceResults | person, friend
|      0 |   1 |      0 |           |           |           |
|          |
|          |
+-----+
+-----+-----+-----+-----+
| +Filter | cache[r.metIn] ENDS WITH $autostring_0
|      0 |   1 |      0 |           |           |           |
|          |
|          |
+-----+
+-----+-----+-----+-----+
| +DirectedRelationshipIndexScan | RANGE INDEX (person)-[r:KNOWS(metIn, lastMetIn)]->(friend) WHERE metIn
IS NOT NULL AND lastMetIn IS |           1 |   1 |      2 |      120 |           2/1 |
0.317 | Fused in Pipeline 0 |
|          | NOT NULL, cache[r.metIn]
|          |           |           |           |           |
|          |
+-----+
+-----+-----+-----+-----+
+-----+
```

Total database accesses: 2, total allocated memory: 184

Substring search using `CONTAINS` (single-property index)

The `CONTAINS` predicate on `person.firstname` in the following query will use the `Person(firstname)` index, if it exists. Text indexes are optimized for `CONTAINS` and `ENDS WITH` and they are the only indexes that can solve those predicates. Composite indexes are currently not able to support `CONTAINS`.

Query

```
MATCH (person:Person)
WHERE person.firstname CONTAINS 'h'
RETURN person
```

Text indexes only index String values and therefore do not find other values.

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

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128
+-----+
+-----+
+-----+-----+-----+-----+
+-----+
| Operator      | Details
| Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline
|               |
+-----+
+-----+
+-----+-----+-----+-----+
+-----+
| +ProduceResults | person
|           15 |   1 |     0 |           |           |           |
|               |
|               |
+-----+
+-----+
| +Filter      | cache[person.country] CONTAINS $autostring_0
|           15 |   1 |     0 |           |           |           |
|               |
|               |
+-----+
+-----+
| +NodeIndexScan | RANGE INDEX person:Person(country, age) WHERE country IS NOT NULL AND age IS NOT NULL,
| cache[person. |       303 |   303 |     304 |       120 |       5/0 |     2.309 |
| Fused in Pipeline 0 |
|           | country |
|           |           |           |           |           |           |
|           |
+-----+
+-----+
+-----+-----+-----+-----+
+-----+
Total database accesses: 304, total allocated memory: 184
```

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

Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

Operator	Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline
+ProduceResults	1	1	0	person, friend			
+DirectedRelationshipIndexScan	1	1	2	RANGE INDEX (person)-[r:KNOWS(since)]->(friend) WHERE since IS NOT NULL	120	2/1	1.046 Fused in Pipeline 0

Total database accesses: 2, total allocated memory: 184

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

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

+-----+
+-----+
+-----+-----+-----+-----+
| Operator      | Details
| Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline
|-----|
+-----+
+-----+-----+-----+-----+
| +ProduceResults | p
|   1 |   2 |     0 |           |           |           |
|   |   |           |           |           |
|   |   |
+-----+
+-----+-----+-----+-----+
| +NodeIndexScan | RANGE INDEX p:Person(firstname, surname) WHERE firstname IS NOT NULL AND surname IS
NOT NULL |           1 |   2 |     3 |       120 |           2/1 |    0.310 | Fused
in Pipeline 0 |
+-----+
+-----+-----+-----+-----+
+-----+-----+-----+-----+
Total database accesses: 3, total allocated memory: 184
```

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

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

+-----+
+-----+
+-----+-----+-----+-----+-----+
| Operator | Details
| Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline
|-----+-----+-----+-----+-----+
| +ProduceResults | `r.lastMetPoint` | | | | |
|   13 | 9 | 0 |           |           |           |
|   |   |   |           |           |           |
|-----+-----+-----+-----+-----+
| +Projection | cache[r.lastMetPoint] AS `r.lastMetPoint` | | | | |
|   13 | 9 | 0 |           |           |           |
|   |   |   |           |           |           |
|-----+-----+-----+-----+-----+
| +Filter | point.distance(cache[r.lastMetPoint], point({x: $autoint_0, y: $autoint_1})) < $autoint_2 | | | |
|           |           13 | 9 | 0 |           |
|           |           |           |           |
|-----+-----+-----+-----+-----+
| +DirectedRelationshipIndexSeekByRange | POINT INDEX (anon_0)-[r:KNOWS(lastMetPoint)]->(anon_1) WHERE
point.distance(lastMetPoint, point($aut |           13 | 9 | 10 | 120 |
5/3 | 1.417 | Fused in Pipeline 0 |           |           |           |
|           |           |           |           |           |           |
|           |           |           |           |           |           |
|-----+-----+-----+-----+-----+
Total database accesses: 10, total allocated memory: 184
```

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

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

+-----+
+-----+
+-----+ +-----+ +-----+ +-----+
+-----+ | Operator | Details
| Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline
+-----+
+-----+
+-----+ +-----+ +-----+ +-----+
+-----+ | +ProduceResults | `person.firstname` |
| 0 | 1 | 0 | |
| |
| |
+-----+
+-----+ +-----+ +-----+ +-----+
| +Projection | person.firstname AS `person.firstname` |
| 0 | 1 | 2 | |
| |
| |
+-----+
+-----+ +-----+ +-----+ +-----+
| +NodeIndexSeekByRange | POINT INDEX person:Person(location) WHERE point.withinBBox(location,
point($autodouble_0, $autodoub | 0 | 1 | 2 | 120 |
6/0 | 7.910 | Fused in Pipeline 0 |
| e_1), point($autodouble_2, $autodouble_3)) |
| |
+-----+
+-----+ +-----+ +-----+ +-----+
+-----+
```

Total database accesses: 4, total allocated memory: 184

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
```


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
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:

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. Louden 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 → server.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 | 72 |
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 5.4.0, 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 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.
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
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:

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. Louden 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 → server.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
Estimate 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 (Bytes)	Details	Estimated Rows	Rows	DB Hits	Memory
Page Cache Hits/Misses	Time (ms)	Other			
+ProduceResults@neo4j	name, count 0/0 0.536 In Pipeline 1	13	102	0	
+EagerAggregation@neo4j	p.name AS name, count(m) AS count 17296 Fused in Pipeline 0	13	102	344	
+Filter@neo4j	p:Person Fused in Pipeline 0	172	172	172	
+Expand(All)@neo4j	(m)->-[anon_16:ACTED_IN]-(p) Fused in Pipeline 0	172	172	254	
+NodeByLabelScan@neo4j	m:Movie 72 5/0 12.818 Fused in Pipeline 0	38	38	39	

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
+ProduceResults@neo4j	numberOfNames	1	1
+EagerAggregation@neo4j	count(DISTINCT cache[p.name]) AS numberOfNames	1	1
+NodeIndexScan@neo4j	p:Person(name) WHERE name IS NOT NULL, cache[p.name]	125	125

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 in 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 5.4.0, 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 three types of planner hints:

- Index hints.
- Scan hints.
- Join hints.

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})
RETURN *
```

The query above will be used in some of the examples on this page. Without any hints, one index and no join is used.

Query plan

Planner COST					
Runtime PIPELINED					
Runtime version 5.4					
Batch size 128					
Operator	Details	Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)
					Pipeline
+ProduceResults	c, cc, i, p, s, sc	0			
+Filter	s.born = \$autoint_0 AND s:Scientist	0			
+Expand(All)	(sc)<-[anon_0:RESEARCHED]-(s)	0			
+Filter	i.year = \$autoint_1 AND sc:Science	0			
+Expand(All)	(p)-[i:INVENTED_BY]->(sc)	0			
+Filter	p.born = \$autoint_2 AND p:Pioneer	2			
+Expand(All)	(c)<-[anon_1:LIVES_IN]-(p)	3			
+Filter	c:City	2			
+Expand(All)	(cc)<-[anon_2:PART_OF]-(c)	2			
+NodeIndexSeek	RANGE INDEX cc:Country(formed) WHERE formed = \$autoint_3	2	120	6/1	0.506 Fused in Pipeline 0
Total database accesses: 11, total allocated memory: 208					

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 [RANGE TEXT POINT] INDEX variable:Label(property)</code>	<code>NodeIndexScan, NodeIndexSeek</code>
<code>USING [RANGE TEXT POINT] INDEX SEEK variable:Label(property)</code>	<code>NodeIndexSeek</code>
<code>USING [RANGE TEXT POINT] INDEX variable:RELATIONSHIP_TYPE(property)</code>	<code>DirectedRelationshipIndexScan, UndirectedRelationshipIndexScan, DirectedRelationshipIndexSeek, UndirectedRelationshipIndexSeek</code>
<code>USING [RANGE TEXT POINT] INDEX SEEK variable:RELATIONSHIP_TYPE(property)</code>	<code>DirectedRelationshipIndexSeek, UndirectedRelationshipIndexSeek</code>

When specifying an index type for a hint, e.g. `RANGE`, `TEXT`, or `POINT`, 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

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline	Estimated Rows	Rows	DB Hits
+ProduceResults	c, cc, i, p, s, sc					0	0	0
+Filter	cc.formed = \$autoint_3 AND cc:Country					0	0	0
+Expand(All)	(c)-[anon_2:PART_OF]->(cc)					0	0	0
+Filter	c:City					0	0	0
+Expand(All)	(p)-[anon_1:LIVES_IN]->(c)					0	0	0
+Filter	s.born = \$autoint_0 AND s:Scientist					0	0	0
+Expand(All)	(sc)<-[anon_0:RESEARCHED]-(s)					0	0	0
+Filter	i.year = \$autoint_1 AND sc:Science					0	0	2
+Expand(All)	(p)-[i:INVENTED_BY]->(sc)					2	2	6
+NodeIndexSeek	RANGE INDEX p:Pioneer(born) WHERE born = \$autoint_2 120 4/1 0.491 Fused in Pipeline 0					2	2	3

Total database accesses: 11, total allocated memory: 208

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

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

+-----+-----+-----+
+-----+-----+-----+
| Operator      | Details          | Estimated Rows | Rows | DB
Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
| +ProduceResults | c               |           |           |           |
0   |                 |           |           |           |           |
| |               +-----+
+-----+-----+-----+-----+
| +NodeIndexSeek | TEXT INDEX c:Country(name) WHERE name = $autostring_0 |
2   |           120 |           2/0 |       0.949 | Fused in Pipeline 0 |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
Total database accesses: 2, total allocated memory: 184
```

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)<-[i: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

Planner COST						
Runtime PIPELINED						
Runtime version 5.4						
Batch size 128						
+-----+ +-----+ +-----+ Operator Details Estimate Rows Rows DB Hits Memory (Bytes) Page Cache Hits/Misses Time (ms) Pipeline						
+-----+ +-----+ +-----+ +ProduceResults c, cc, i, p, s, sc 0 0 0 +-----+ +-----+ +Filter cc.formed = \$autoint_3 AND cc:Country 0 0 0 +-----+ +-----+ +Expand(All) (c)-[anon_2:PART_OF]->(cc) 0 0 0 +-----+ +-----+ +Filter c:City 0 0 0 +-----+ +-----+ +Expand(All) (p)-[anon_1:LIVES_IN]->(c) 0 0 0 +-----+ +-----+ +Filter s.born = \$autoint_0 AND s:Scientist 0 0 0 +-----+ +-----+ +Expand(All) (sc)<-[anon_0:RESEARCHED]-(s) 0 0 0 +-----+ +-----+ +Filter p.born = \$autoint_2 AND sc:Science AND p:Pioneer 0 0 4 +-----+ +-----+ +DirectedRelationshipIndexSeek RANGE INDEX (p)-[i:INVENTED_BY(year)]->(sc) WHERE year = \$autoint_1 2 2 3 120 5/1 0.461 Fused in Pipeline 0 +-----+ +-----+ +-----+						
Total database accesses: 7, total allocated memory: 208						

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

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

+-----+
+-----+-----+-----+-----+-----+
| Operator | Details
| Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline
|           |       |       |       |       |       |
+-----+-----+-----+-----+-----+-----+
| +ProduceResults | i
|   1 | 1 | 0 |       |       |       |
|   |   |   |       |       |       |
|   |   |   |       |       |       |
+-----+-----+-----+-----+-----+
| +DirectedRelationshipIndexSeek | TEXT INDEX (anon_0)-[i:INVENTED_BY(location)]->(anon_1) WHERE location
= $autostring_0 |           1 | 1 | 2 | 120 | 3/0 | 1.079 |
Fused in Pipeline 0 |
+-----+
+-----+-----+-----+-----+-----+
Total database accesses: 2, total allocated memory: 184
```

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)<-[: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

Planner COST					
Runtime PIPELINED					
Runtime version 5.4					
Batch size 128					
<pre>+-----+-----+-----+-----+-----+ Operator Details Estimated Rows Rows DB Hits Memory (Bytes) Page Cache Hits/Misses Time (ms) Pipeline +-----+-----+-----+-----+-----+ +ProduceResults c, cc, i, p, s, sc 0 0 0 0 0/0 0.000 +-----+-----+-----+-----+ +NodeHashJoin sc 0 0 0 0 432 In Pipeline 2 \ +-----+-----+-----+-----+ +Expand(All) (s)-[anon_0:RESEARCHED]->(sc) 1 0 0 0 +-----+-----+-----+-----+ +NodeIndexSeek RANGE INDEX s:Scientist(born) WHERE born = \$autoint_0 1 0 0 0 120 0/0 0.000 Fused in Pipeline 1 +-----+-----+-----+-----+ +Filter i.year = \$autoint_1 AND sc:Science 0 0 0 0 +-----+-----+-----+-----+ +Expand(All) (p)-[i:INVENTED_BY]->(sc) 0 0 0 0 +-----+-----+-----+-----+ +Filter p.born = \$autoint_2 AND p:Pioneer 0 0 0 2 +-----+-----+-----+-----+ +Expand(All) (c)<-[anon_1:LIVES_IN]-(p) 1 1 1 3 +-----+-----+-----+-----+ +Filter c:City 1 1 1 2 +-----+-----+-----+-----+ +Expand(All) (cc)<-[anon_2:PART_OF]-(c) 1 1 1 2 +-----+-----+-----+-----+ +NodeIndexSeek RANGE INDEX cc:Country(formed) WHERE formed = \$autoint_3 1 1 1 2 120 7/0 0.494 Fused in Pipeline 0 +-----+-----+-----+-----+-----+</pre>					
Total database accesses: 11, total allocated memory: 768					

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(formatted)
WHERE country.formed = 500 OR country.name STARTS WITH "A"
RETURN *
```

Query plan

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

+-----+
+-----+-----+-----+-----+-----+-----+
| Operator      | Details
| Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline
|               |
+-----+-----+-----+-----+-----+-----+
| +ProduceResults | country
| 1 | 1 | 0 |           |           |           |           |
| | | +-----+
+-----+-----+-----+-----+-----+-----+
| +Distinct      | country
| 1 | 1 | 0 |           | 224 |           |           |
| | | +-----+
+-----+-----+-----+-----+-----+-----+
| +Union          |
| 2 | 1 | 0 |           | 80 |           | 1/0 | 0.213 | Fused in Pipeline 2 |
| | \ +-----+
+-----+-----+-----+-----+-----+-----+
| +NodeIndexSeek   | RANGE INDEX country:Country(formatted) WHERE formed = $autoint_0
| 1 | 1 | 2 |           | 120 |           | 1/0 | 0.101 | In Pipeline 1 |
| | +-----+
+-----+-----+-----+-----+-----+-----+
| +NodeIndexSeekByRange | RANGE INDEX country:Country(name) WHERE name STARTS WITH $autostring_1
| 1 | 0 | 1 |           | 120 |           | 0/1 | 0.307 | In Pipeline 0 |
+-----+-----+-----+-----+-----+-----+
Total database accesses: 3, total allocated memory: 320
```

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)<-[i: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

Planner COST					
Runtime PIPELINED					
Runtime version 5.4					
Batch size 128					
Operator Details					
DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline	Estimated Rows Rows
+ProduceResults	c, cc, i, p, s, sc				0 0
0					
+Filter	cc.formed = \$autoint_3 AND cc:Country				0 0
0					
+Expand(All)	(c)-[anon_2:PART_OF]->(cc)				0 0
0					
+Filter	c:City				0 0
0					
+Expand(All)	(p)-[anon_1:LIVES_IN]->(c)				0 0
0					
+Filter	i.year = \$autoint_1 AND p.born = \$autoint_2 AND p:Pioneer				0 0
1					
+Expand(All)	(sc)<-[i:INVENTED_BY]-(p)				1 1
3					
+Filter	sc:Science				1 1
2					
+Expand(All)	(s)-[anon_0:RESEARCHED]->(sc)				1 1
2					
+Filter	s.born = \$autoint_0				1 1
200					
+NodeByLabelScan	s:Scientist				100 100
101	120	11/0	0.512	Fused in Pipeline 0	

Total database accesses: 309, total allocated memory: 216

Hinting a relationship type scan

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 SCAN i:INVENTED_BY
RETURN *
```

Query plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator Details						
Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline
+ProduceResults	0	0	c, cc, i, p, s, sc	0		
+Filter	0	0	cc.formed = \$autoint_3 AND cc:Country	0		
+Expand(All)	0	0	(c)-[anon_2:PART_OF]->(cc)	0		
+Filter	0	0	c:City	0		
+Expand(All)	0	0	(p)-[anon_1:LIVES_IN]->(c)	0		
+Filter	0	0	s.born = \$autoint_0 AND s:Scientist	0		
+Expand(All)	0	0	(sc)<-[anon_0:RESEARCHED]-(s)	0		
+Filter	0	0	i.year = \$autoint_1 AND p.born = \$autoint_2 AND sc:Science AND p:Pioneer	204		
+DirectedRelationshipTypeScan	100	100	(p)-[i:INVENTED_BY]->(sc)	101	120	
Pipeline 0					9/0	0.910 Fused in

Total database accesses: 305, total allocated memory: 208

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 [UnionNodeByLabelsScan](#).

Query

```
MATCH (person)
USING SCAN person:Pioneer
USING SCAN person:Scientist
WHERE person:Pioneer OR person:Scientist
RETURN *
```

Query plan

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

+-----+-----+-----+-----+
| Operator | Details | Estimated Rows | Rows | DB Hits | Memory (Bytes) |
Page Cache Hits/Misses | Time (ms) | Pipeline |
+-----+-----+-----+-----+
+-----+-----+-----+
| +ProduceResults | person | 180 | 200 | 0 | |
| | | +-----+-----+-----+-----+
| | | +-----+-----+-----+-----+
| | | +UnionNodeByLabelsScan | person:Pioneer|Scientist | 180 | 200 | 202 | 120 |
6/0 | 1.740 | Fused in Pipeline 0 |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
Total database accesses: 202, total allocated memory: 184
```

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)<-[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)
USING JOIN ON p
RETURN *
```

Query plan

Planner COST																																																																																			
Runtime PIPELINED																																																																																			
Runtime version 5.4																																																																																			
Batch size 128																																																																																			
<table border="1"><thead><tr><th>Operator</th><th>Details</th><th>Estimated Rows</th></tr></thead><tbody><tr><td>Rows</td><td>DB Hits Memory (Bytes) Page Cache Hits/Misses Time (ms)</td><td></td></tr><tr><td>+ProduceResults</td><td>c, cc, i, p, s, sc</td><td>0</td></tr><tr><td>0</td><td>0 0/0 0.000 </td><td></td></tr><tr><td>+NodeHashJoin</td><td>p</td><td>0</td></tr><tr><td>0</td><td>0 432 </td><td>In Pipeline 2</td></tr><tr><td>+Filter</td><td>cache[p.born] = \$autoint_2</td><td>1</td></tr><tr><td>0</td><td>0 </td><td></td></tr><tr><td>+Expand(All)</td><td>(c)<-[anon_1:LIVES_IN]-(p)</td><td>1</td></tr><tr><td>0</td><td>0 </td><td></td></tr><tr><td>+Filter</td><td>c:City</td><td>1</td></tr><tr><td>0</td><td>0 </td><td></td></tr><tr><td>+Expand(All)</td><td>(cc)<-[anon_2:PART_OF]-(c)</td><td>1</td></tr><tr><td>0</td><td>0 </td><td></td></tr><tr><td>+NodeIndexSeek</td><td>RANGE INDEX cc:Country(formed) WHERE formed = \$autoint_3</td><td>1</td></tr><tr><td>0</td><td>0 120 0/0 0.000 Fused in Pipeline 1</td><td></td></tr><tr><td>+Filter</td><td>i.year = \$autoint_1 AND cache[p.born] = \$autoint_2 AND p:Pioneer</td><td>0</td></tr><tr><td>0</td><td>1 </td><td></td></tr><tr><td>+Expand(All)</td><td>(sc)<-[i:INVENTED_BY]-(p)</td><td>1</td></tr><tr><td>1</td><td>3 </td><td></td></tr><tr><td>+Filter</td><td>sc:Science</td><td>1</td></tr><tr><td>1</td><td>2 </td><td></td></tr><tr><td>+Expand(All)</td><td>(s)-[anon_0:RESEARCHED]->(sc)</td><td>1</td></tr><tr><td>1</td><td>2 </td><td></td></tr><tr><td>+NodeIndexSeek</td><td>RANGE INDEX s:Scientist(born) WHERE born = \$autoint_0</td><td>1</td></tr><tr><td>1</td><td>2 120 6/1 0.515 Fused in Pipeline 0</td><td></td></tr></tbody></table>						Operator	Details	Estimated Rows	Rows	DB Hits Memory (Bytes) Page Cache Hits/Misses Time (ms)		+ProduceResults	c, cc, i, p, s, sc	0	0	0 0/0 0.000		+NodeHashJoin	p	0	0	0 432	In Pipeline 2	+Filter	cache[p.born] = \$autoint_2	1	0	0		+Expand(All)	(c)<-[anon_1:LIVES_IN]-(p)	1	0	0		+Filter	c:City	1	0	0		+Expand(All)	(cc)<-[anon_2:PART_OF]-(c)	1	0	0		+NodeIndexSeek	RANGE INDEX cc:Country(formed) WHERE formed = \$autoint_3	1	0	0 120 0/0 0.000 Fused in Pipeline 1		+Filter	i.year = \$autoint_1 AND cache[p.born] = \$autoint_2 AND p:Pioneer	0	0	1		+Expand(All)	(sc)<-[i:INVENTED_BY]-(p)	1	1	3		+Filter	sc:Science	1	1	2		+Expand(All)	(s)-[anon_0:RESEARCHED]->(sc)	1	1	2		+NodeIndexSeek	RANGE INDEX s:Scientist(born) WHERE born = \$autoint_0	1	1	2 120 6/1 0.515 Fused in Pipeline 0	
Operator	Details	Estimated Rows																																																																																	
Rows	DB Hits Memory (Bytes) Page Cache Hits/Misses Time (ms)																																																																																		
+ProduceResults	c, cc, i, p, s, sc	0																																																																																	
0	0 0/0 0.000																																																																																		
+NodeHashJoin	p	0																																																																																	
0	0 432	In Pipeline 2																																																																																	
+Filter	cache[p.born] = \$autoint_2	1																																																																																	
0	0																																																																																		
+Expand(All)	(c)<-[anon_1:LIVES_IN]-(p)	1																																																																																	
0	0																																																																																		
+Filter	c:City	1																																																																																	
0	0																																																																																		
+Expand(All)	(cc)<-[anon_2:PART_OF]-(c)	1																																																																																	
0	0																																																																																		
+NodeIndexSeek	RANGE INDEX cc:Country(formed) WHERE formed = \$autoint_3	1																																																																																	
0	0 120 0/0 0.000 Fused in Pipeline 1																																																																																		
+Filter	i.year = \$autoint_1 AND cache[p.born] = \$autoint_2 AND p:Pioneer	0																																																																																	
0	1																																																																																		
+Expand(All)	(sc)<-[i:INVENTED_BY]-(p)	1																																																																																	
1	3																																																																																		
+Filter	sc:Science	1																																																																																	
1	2																																																																																		
+Expand(All)	(s)-[anon_0:RESEARCHED]->(sc)	1																																																																																	
1	2																																																																																		
+NodeIndexSeek	RANGE INDEX s:Scientist(born) WHERE born = \$autoint_0	1																																																																																	
1	2 120 6/1 0.515 Fused in Pipeline 0																																																																																		
Total database accesses: 10, total allocated memory: 768																																																																																			

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

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

+-----+-----+-----+
| Operator      | Details          | Estimated Rows | Rows |
+-----+-----+-----+
| DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline
+-----+-----+-----+
| +ProduceResults | s, sc           |           |           | 1 | 1 |
0 |           |           |           |           |   |
| |           +-----+
+-----+-----+-----+
| +OptionalExpand(All) | (s)-[anon_0:RESEARCHED]->(sc) WHERE sc:Science |           |           | 1 | 1 |
4 |           |           |           |           |   |
| |           +-----+
+-----+-----+-----+
| +NodeIndexSeek | RANGE INDEX s:Scientist(born) WHERE born = $autoint_0 |           |           | 1 | 1 |
2 |           120 |           6/0 |           0.560 | Fused in Pipeline 0 |
+-----+-----+-----+
+-----+-----+-----+
Total database accesses: 6, total allocated memory: 184
```

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

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details				Estimated Rows	Rows		
		DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline		
+ProduceResults	s, sc	0	2/0	0.213			1	1
+NodeLeftOuterHashJoin	s	0	3112		0.650	In Pipeline 2	1	1
+Expand(All)	(sc)<-[anon_0:RESEARCHED]->(s)	300					100	100
+NodeByLabelScan	sc:Science	101	120	4/0	0.786	Fused in Pipeline 1	100	100
+NodeIndexSeek	RANGE INDEX s:Scientist(born) WHERE born = \$autoint_0	2	120	1/0	0.214	In Pipeline 0	1	1

Total database accesses: 403, total allocated memory: 3192

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.

IDs

Each operator is assigned a unique ID, which is shown in the execution plan. The IDs can be used to refer unambiguously to operators. There are no guarantees about the order of IDs, although they will usually start with 0 at the root, and will increase towards the leaves of the tree.

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
AllNodesScan	Reads all nodes from the node store.	Yes		
Anti	Tests for the absence of a pattern.			
AntiSemiApply	Performs a nested loop. Tests for the absence of a pattern predicate.			
Apply	Performs a nested loop. Yields rows from both the left-hand and right-hand side operators.			

Name	Description	Leaf?	Updating?	Considerations
Argument	Indicates the variable to be used as an argument to the right-hand side of an Apply operator.	Yes		
AssertSameNode	Used to ensure that no property uniqueness constraints are violated.			
AssertingMultiNodeIndexSeek	Used to ensure that no property uniqueness constraints are violated.			
CacheProperties	Reads node or relationship properties and caches them.			
CartesianProduct	Produces a cartesian product of the inputs from the left-hand and right-hand operators.			
Create	Creates nodes and relationships.	Yes		
CreateIndex	Creates an index for either nodes or relationships.	Yes		
CreateConstraint	Creates a constraint for either nodes or relationships.	Yes		
Delete	Deletes a node or relationship.	Yes		
DetachDelete	Deletes a node and its relationships.	Yes		
DirectedAllRelationshipsScan	Fetches all relationships and their start and end nodes in the database.			
DirectedRelationshipByIdSeek	Reads one or more relationships by id from the relationship store.	Yes		

Name	Description	Leaf?	Updating?	Considerations
DirectedRelationshipIndexContainsScan	Examines all values stored in an index, searching for entries containing a specific string; for example, in queries including CONTAINS .			
DirectedRelationshipIndexEndsWithScan	Examines all values stored in an index, searching for entries ending in a specific string; for example, in queries containing ENDS WITH .			
DirectedRelationshipIndexScan	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.			
DirectedRelationshipIndexSeek	Finds relationships and their start and end nodes using an index seek.			
DirectedRelationshipIndexSeekByRange	Finds relationships and their start and end nodes using an index seek where the value of the property matches a given prefix string.			
DirectedRelationshipTypeScan	Fetches all relationships and their start and end nodes with a specific type from the relationship type index.			
DirectedUnionRelationshipTypesScan	Fetches all relationships and their start and end nodes with at least one of the provided types from the relationship type index.			
Distinct	Drops duplicate rows from the incoming stream of rows.			Eager
DoNothingIfExists(CONSTRAINT)	Checks if a constraint already exists, if it does then it stops the execution, if not it continues.	Yes		

Name	Description	Leaf?	Updating?	Considerations
DoNothingIfExists(INDEX)	Checks if an index already exists, if it does then it stops the execution, if not it continues.	Yes		
DropConstraint	Drops a constraint using its name.	Yes	Yes	
DropIndex	Drops an index using its name.	Yes	Yes	
Eager	For isolation purposes, Eager 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 ExhaustiveLimit operator is similar to the Limit operator, but always exhausts the input. Used when combining LIMIT and updates.			
Expand(All)	Traverses incoming or outgoing relationships from a given node.			
Expand(Into)	Finds all relationships between two nodes.			
Filter	Filters each row coming from the child operator, only passing through rows that evaluate the predicates to true.			

Name	Description	Leaf?	Updating?	Considerations
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 <code>n</code> rows from the incoming input.			
LoadCSV	Loads data from a CSV source into the query.	Yes		
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		

Name	Description	Leaf?	Updating?	Considerations
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		
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		

Name	Description	Leaf?	Updating?	Considerations
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.			
PartialTop	Returns the first <code>n</code> 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.			

Name	Description	Leaf?	Updating?	Considerations
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 <code>false</code> .			
SelectOrSemiApply	Performs a nested loop. Tests for the presence of a pattern predicate if an expression predicate evaluates to <code>false</code> .			
SemiApply	Performs a nested loop. Tests for the presence of a pattern predicate.			
SetLabels	Sets labels on a node.	Yes		
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.			

Name	Description	Leaf?	Updating?	Considerations
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 ' <i>n</i> ' rows sorted by a provided key.			Eager
TriadicBuild	The TriadicBuild operator is used in conjunction with TriadicFilter to solve triangular queries.			
TriadicFilter	The TriadicFilter operator is used in conjunction with TriadicBuild to solve triangular queries.			
TriadicSelection	Solves triangular queries, such as the very common 'find my friend-of-friends that are not already my friend'.			
UndirectedAllRelationshipsScan	Fetches all relationships and their start and end nodes in the database.			

Name	Description	Leaf?	Updating?	Considerations
UndirectedRelationshipByIdSeek	Reads one or more relationships by ID from the relationship store.	Yes		
UndirectedRelationshipIndexContainsScan	Examines all values stored in an index, searching for entries containing a specific string; for example, in queries including CONTAINS.			
UndirectedRelationshipIndexEndsWithScan	Examines all values stored in an index, searching for entries ending in a specific string; for example, in queries containing ENDS WITH.			
UndirectedRelationshipIndexScan	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.			
UndirectedRelationshipIndexSeek	Finds relationships and their start and end nodes using an index seek.			
UndirectedRelationshipIndexSeekByRange	Finds relationships and their start and end nodes using an index seek where the value of the property matches a given prefix string.			
UndirectedRelationshipTypeScan	Fetches all relationships and their start and end nodes with a specific type from the relationship type index.			
UndirectedUnionRelationshipTypesScan	Fetches all relationships and their start and end nodes with at least one of the provided types from the relationship type index.			
Union	Concatenates the results from the right-hand operator with the results from the left-hand operator.			

Name	Description	Leaf?	Updating?	Considerations
UnionNodeByLabelsScan	Fetches all nodes that have at least one of the provided labels from the node label index.			
Unwind	Returns one row per item in a list.			
ValueHashJoin	Executes a hash join on arbitrary values.			Eager
VarLengthExpand(All)	Traverses variable-length relationships from a given node.			
VarLengthExpand(Into)	Finds all variable-length relationships between two nodes.			
VarLengthExpand(Pruning)	Traverses variable-length relationships from a given node and only returns unique end nodes.			
VarLengthExpand(Pruning,BFS)	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 415. AllNodesScan

Query

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

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline
+ProduceResults	n	35	35	0				
+AllNodesScan	n	35	35	36	120			
3/0	0.354	Fused in Pipeline 0						

Total database accesses: 36, total allocated memory: 184

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 416. DirectedRelationshipIndexScan

Query

```
PROFILE
MATCH ()-[r: WORKS_IN]->()
WHERE r.title IS NOT NULL
RETURN r
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator Details						
Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline
+ProduceResults	r					
15	15	0				
+DirectedRelationshipIndexScan	RANGE INDEX (anon_0)-[r:WORKS_IN(title)]->(anon_1) WHERE title IS NOT NULL	15	15	16	120	3/1 2.464
Fused in Pipeline 0						
Total database accesses: 16, total allocated memory: 184						

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 417. UndirectedRelationshipIndexScan

Query

```
PROFILE
MATCH ()-[r: WORKS_IN]-()
WHERE r.title IS NOT NULL
RETURN r
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator		Details				
Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline
+ProduceResults	30	30	0	r		
+UndirectedRelationshipIndexScan	30	30	16	120	3/1	1.266
Fused in Pipeline 0						

Total database accesses: 16, total allocated memory: 184

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 418. DirectedRelationshipIndexSeek

Query

```
PROFILE
MATCH (candidate)-[r:WORKS_IN]->()
WHERE r.title = 'chief architect'
RETURN candidate
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator		Details				
Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline
<hr/>						
+ProduceResults						
<hr/>						
+DirectedRelationshipIndexSeek						
RANGE INDEX (candidate)-[r:WORKS_IN(title)]->(anon_0) WHERE title = \$autostring_0	2	1	2	120	3/1	
0.591 Fused in Pipeline 0						
<hr/>						
<hr/>						
Total database accesses: 2, total allocated memory: 184						

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 419. UndirectedRelationshipIndexSeek

Query

```
PROFILE
MATCH (candidate)-[r:WORKS_IN]-()
WHERE r.title = 'chief architect'
RETURN candidate
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator Details						
Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline
+ProduceResults	candidate					
4	2	0				
+UndirectedRelationshipIndexSeek	RANGE INDEX (candidate)-[r:WORKS_IN(title)]-(anon_0) WHERE title = \$autostring_0	4	2	2	120	3/1
0.791	Fused in Pipeline 0					

Total database accesses: 2, total allocated memory: 184

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 420. DirectedRelationshipByIdSeek

Query

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

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows	Rows		
DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline	
+ProduceResults	r, n1				1 1
0					
+DirectedRelationshipByIdSeek (n1)-[r]->(anon_0) WHERE id(r) = \$autoint_0	120 4/0 0.459 Fused in Pipeline 0			1 1	

Total database accesses: 1, total allocated memory: 184

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 421. UndirectedRelationshipByIdSeek

Query

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

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator		Details				Estimated Rows	
Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline		
2	0		r, n1				2
2	1	120		4/0	0.332	Fused in Pipeline 0	2
Total database accesses: 1, total allocated memory: 184							

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 422. DirectedRelationshipIndexContainsScan

Query

```
PROFILE
MATCH ()-[r: WORKS_IN]->()
WHERE r.title CONTAINS 'senior'
RETURN r
```

Query Plan

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128
+-----+
+-----+
+-----+-----+-----+-----+
| Operator | Details
| Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline
|           |
+-----+
+-----+-----+-----+-----+
| +ProduceResults | r
|     0 |    4 |     0 |           |           |           |
|           |
|           |
+-----+-----+-----+-----+
| +DirectedRelationshipIndexContainsScan | TEXT INDEX (anon_0)-[r:WORKS_IN(title)]->(anon_1) WHERE
title CONTAINS $autostring_0 |          0 |    4 |     5 |      120 |
3/0 |   1.051 | Fused in Pipeline 0 |
+-----+
+-----+-----+-----+-----+
+-----+
Total database accesses: 5, total allocated memory: 184
```

Undirected Relationship Index Contains Scan

The [UndirectedRelationshipIndexContainsScan](#) operator examines all values stored in an index, searching for entries containing a specific string; for example, in queries including `CONTAINS`. Although this is slower than an index seek (since all entries need to be examined), it is still faster than the indirection resulting from a type scan using [DirectedRelationshipTypeScan](#), and a property store filter.

Example 423. UndirectedRelationshipIndexContainsScan

Query

```
PROFILE
MATCH ()-[r: WORKS_IN]-()
WHERE r.title CONTAINS 'senior'
RETURN r
```

Query Plan

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128
+-----+
+-----+
+-----+-----+-----+-----+
+-----+
| Operator | Details
| Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline
|           |
+-----+
+-----+
+-----+-----+-----+-----+
+-----+
| +ProduceResults | r
|     0 |    8 |     0 |           |           |           |
|           |
|           |
+-----+
+-----+
+-----+-----+-----+-----+
+-----+
| +UndirectedRelationshipIndexContainsScan | TEXT INDEX (anon_0)-[r:WORKS_IN(title)]-(anon_1) WHERE
title CONTAINS $autostring_0 |          0 |    8 |     5 |      120 |
3/0 |    2.684 | Fused in Pipeline 0 |
+-----+
+-----+
+-----+-----+-----+-----+
+-----+
Total database accesses: 5, total allocated memory: 184
```

Directed Relationship Index Ends With Scan

The [DirectedRelationshipIndexEndsWithScan](#) operator examines all values stored in an index, searching for entries ending in a specific string; for example, in queries containing [ENDS WITH](#). Although this is slower than an index seek (since all entries need to be examined), it is still faster than the indirection resulting from a label scan using [NodeByLabelScan](#), and a property store filter.

Example 424. DirectedRelationshipIndexEndsWithScan

Query

```
PROFILE
MATCH ()-[r: WORKS_IN]->()
WHERE r.title ENDS WITH 'developer'
RETURN r
```

Query Plan

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

+-----+
+-----+
+-----+-----+-----+-----+
| Operator | Details
| Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline
|           |
+-----+
+-----+-----+-----+-----+
| +ProduceResults | r
|     0 |    8 |     0 |           |           |           |
|           |
|           |
+-----+-----+-----+-----+
| +DirectedRelationshipIndexEndsWithScan | TEXT INDEX (anon_0)-[r:WORKS_IN(title)]->(anon_1) WHERE
title ENDS WITH $autostring_0 |          0 |    8 |     9 |      120 |
3/0 |    1.887 | Fused in Pipeline 0 |
+-----+
+-----+-----+-----+-----+
+-----+
```

Total database accesses: 9, total allocated memory: 184

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 425. UndirectedRelationshipIndexEndsWithScan

Query

```
PROFILE
MATCH ()-[r: WORKS_IN]-()
WHERE r.title ENDS WITH 'developer'
RETURN r
```

Query Plan

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128
+-----+
+-----+
+-----+-----+-----+-----+
+-----+
| Operator | Details
| Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline
|           |
+-----+
+-----+-----+-----+-----+
+-----+
| +ProduceResults | r
|     0 |    16 |      0 |          |           |           |
|           |
|           |
+-----+-----+-----+-----+
+-----+
| +UndirectedRelationshipIndexEndsWithScan | TEXT INDEX (anon_0)-[r:WORKS_IN(title)]-(anon_1) WHERE
title ENDS WITH $autostring_0 |          0 |    16 |      9 |        120 |
3/0 |    1.465 | Fused in Pipeline 0 |
+-----+
+-----+-----+-----+-----+
+-----+
Total database accesses: 9, total allocated memory: 184
```

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 426. DirectedRelationshipIndexSeekByRange

Query

```
PROFILE
MATCH (candidate: Person)-[r:WORKS_IN]->(location)
WHERE r.duration > 100
RETURN candidate
```

Query Plan

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

+-----+
+-----+
+-----+-----+-----+-----+
| Operator | Details
| Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline
|           |
+-----+
+-----+
+-----+-----+-----+-----+
| +ProduceResults | candidate
|     4 |   15 |     0 |           |
|           |           |
|           |
|           |
+-----+
+-----+
| +Filter | candidate:Person
|     4 |   15 |    30 |           |
|           |           |
|           |
+-----+
+-----+
| +DirectedRelationshipIndexSeekByRange | RANGE INDEX (candidate)-[r:WORKS_IN(duration)]->(location)
WHERE duration > $autoint_0 |     4 |   15 |    16 |      120 |
4/1 |   0.703 | Fused in Pipeline 0 |
+-----+
+-----+
+-----+-----+-----+-----+
Total database accesses: 46, total allocated memory: 184
```

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 427. UndirectedRelationshipIndexSeekByRange

Query

```
PROFILE
MATCH (candidate: Person)-[r:WORKS_IN]-(location)
WHERE r.duration > 100
RETURN candidate
```

Query Plan

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

+-----+
+-----+
+-----+-----+-----+-----+
| Operator | Details
| Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline
|           |
+-----+
+-----+
+-----+-----+-----+-----+
| +ProduceResults | candidate
|      5 | 15 |     0 |           |
|           |
|           |
+-----+
+-----+
| +Filter | candidate:Person
|      5 | 15 |    60 |           |
|           |
|           |
+-----+
+-----+
| +UndirectedRelationshipIndexSeekByRange | RANGE INDEX (candidate)-[r:WORKS_IN(duration)]-(location)
| WHERE duration > $autoInt_0 |      8 |   30 |     16 |       120 |
| 4/1 | 1.214 | Fused in Pipeline 0 |
+-----+
+-----+
+-----+-----+-----+-----+
Total database accesses: 76, total allocated memory: 184
```

Union Node By Labels Scan

The [UnionNodeByLabelsScan](#) operator fetches all nodes that have at least one of the provided labels from the node label index.

Query

```
PROFILE
MATCH (countryOrLocation:Country|Location)
RETURN countryOrLocation
```

Query Plan

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

+-----+-----+-----+
| Operator | Details | Estimated Rows | Rows | DB Hits | Memory
| (Bytes) | Page Cache Hits/Misses | Time (ms) | Ordered by | Pipeline |
+-----+-----+-----+
| +ProduceResults | countryOrLocation | 17 | 11 | 0 |
| | | |
| | +-----+
| +UnionNodeByLabelsScan | countryOrLocation:Country|Location | 17 | 11 | 13 |
| 120 | 3/1 | 0.660 | countryOrLocation ASC | Fused in Pipeline 0 |
+-----+-----+-----+
+-----+-----+-----+
Total database accesses: 13, total allocated memory: 184
```

Directed All Relationships Scan

The `DirectedAllRelationshipsScan` operator fetches all relationships and their start and end nodes in the database.

Query

```
PROFILE
MATCH ()-[r]->() RETURN r
```

Query Plan

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

+-----+-----+-----+
| Operator | Details | Estimated Rows | Rows | DB Hits | Memory
| (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline |
+-----+-----+-----+
| +ProduceResults | r | 28 | 28 | 0 |
| | | |
| | +-----+
| +DirectedAllRelationshipsScan | (anon_0)-[r]->(anon_1) | 28 | 28 | 28 |
| 120 | 3/0 | 0.502 | Fused in Pipeline 0 |
+-----+-----+-----+
+-----+-----+-----+
Total database accesses: 28, total allocated memory: 184
```

Undirected All Relationships Scan

The [UndirectedAllRelationshipsScan](#) operator fetches all relationships and their start and end nodes in the database.

Query

```
PROFILE  
MATCH ()-[r]-() RETURN r
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator (Bytes)	Details			Estimated Rows	Rows	DB Hits	Memory
	Page Cache Hits/Misses	Time (ms)	Pipeline				
+ProduceResults r 56 56 0							
+UndirectedAllRelationshipsScan (anon_0)-[r]-(anon_1) 56 56 28	120 3/0 1.110 Fused in Pipeline 0						

Total database accesses: 28, total allocated memory: 184

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 428. DirectedRelationshipTypeScan

Query

```
PROFILE  
MATCH ()-[r: FRIENDS_WITH]->()  
RETURN r
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details			Estimated Rows	Rows	DB
	Hits	Memory (Bytes)	Page Cache Hits/Misses			
+ProduceResults	r				12	12
0						

+DirectedRelationshipTypeScan	(anon_0)-[r:FRIENDS_WITH]->(anon_1)				12	12
13	120	2/1	0.557	Fused in Pipeline 0		

Total database accesses: 13, total allocated memory: 184

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

```
PROFILE  
MATCH ()-[r: FRIENDS_WITH]-()  
RETURN r
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details			Estimated Rows	Rows	DB
	Hits	Memory (Bytes)	Page Cache Hits/Misses			
+ProduceResults	r				24	24
+UndirectedRelationshipTypeScan	(anon_0)-[r:FRIENDS_WITH]-(anon_1)	120	2/1	0.749	Fused in Pipeline 0	24

Total database accesses: 13, total allocated memory: 184

Directed Union Relationship Types Scan

The [DirectedUnionRelationshipTypeScan](#) operator fetches all relationships and their start and end nodes with at least one of the provided types from the relationship type index.

Example 429. DirectedUnionRelationshipTypeScan

Query

```
PROFILE
MATCH ()-[friendOrFoe: FRIENDS_WITH|FOE]->()
RETURN friendOrFoe
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Pipeline	Details					Ordered by	Estimated
	Rows	Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses		
+ProduceResults				friendOrFoe			
15 12 0							
+DirectedUnionRelationshipTypesScan				(anon_0)-[friendOrFoe:FRIENDS_WITH FOE]->(anon_1)			
15 12 14			120	3/1	2.027	friendOrFoe ASC	Fused
in Pipeline 0							
Total database accesses: 14, total allocated memory: 184							

Undirected Union Relationship Types Scan

The [UndirectedUnionRelationshipTypeScan](#) operator fetches all relationships and their start and end nodes with at least one of the provided types from the relationship type index.

Example 430. UndirectedUnionRelationshipTypeScan

Query

```
PROFILE
MATCH ()-[friendOrFoe: FRIENDS_WITH|FOE]-()
RETURN friendOrFoe
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator Details						
Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Ordered by
Pipeline						
+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+
+ProduceResults			friendOrFoe			
30 24 0						
+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+
+UndirectedUnionRelationshipTypesScan (anon_0)-[friendOrFoe:FRIENDS_WITH FOE]-(anon_1)						
30 24 14 120 3/1 0.887 friendOrFoe ASC Fused						
in Pipeline 0						
+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+
Total database accesses: 14, total allocated memory: 184						

Node By Id Seek

The [NodeByIdSeek](#) operator reads one or more nodes by id from the node store.

Example 431. NodeByIdSeek

Query

```
PROFILE
MATCH (n)
WHERE elementId(n) = 0
RETURN n
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows	Rows	DB Hits
Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline	
+ProduceResults	n		1	0
+NodeByIdSeek	n WHERE elementId(n) = \$autoInt_0	1	1	1
120	3/0 2.108 Fused in Pipeline 0			

Total database accesses: 1, total allocated memory: 184

Node By Label Scan

The [NodeByLabelScan](#) operator fetches all nodes with a specific label from the node label index.

Example 432. NodeByLabelScan

Query

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

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline
+ProduceResults	person	14	14	0				
+NodeByLabelScan	person:Person	14	14	15	120			

2/1 | 0.522 | Fused in Pipeline 0 |

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 433. NodeIndexSeek

Query

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

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows
Rows	DB Hits Memory (Bytes) Page Cache Hits/Misses Time (ms)	
+ProduceResults	location	1
1 0		
+-----+-----+-----+-----+-----+-----+-----+		
+NodeIndexSeek RANGE INDEX location:Location(name) WHERE name = \$autostring_0 1		
1 2 120 2/1 0.401 Fused in Pipeline 0		

Total database accesses: 2, total allocated memory: 184

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 434. NodeUniqueIndexSeek

Query

```
PROFILE  
MATCH (t:Team {name: 'Malmo'})  
RETURN t
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline	Estimated Rows	Rows	DB
+ProduceResults	t	0					1	0	
+NodeUniqueIndexSeek	UNIQUE t:Team(name) WHERE name = \$autostring_0	1	120	0/1	0.280	Fused in Pipeline 0	1	0	

Total database accesses: 1, total allocated memory: 184

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 435. MultiNodeIndexSeek

Query

```
PROFILE
CYpher runtime=pipelined
MATCH
  (location:Location {name: 'Malmo'}),
  (person:Person {name: 'Bob'})
RETURN location, person
```

Query Plan

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

+-----+
+-----+-----+-----+-----+
| Operator      | Details                                | Estimated |
| Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
| +ProduceResults | location, person
1 |   1 |     0 |           |           |           |           |
| |   +-----+
+-----+-----+-----+-----+
| +MultiNodeIndexSeek | RANGE INDEX location:Location(name) WHERE name = $autostring_0,
1 |   0 |     0 |       120 |        2/2 |    1.910 | Fused in Pipeline 0 |
| |           | RANGE INDEX person:Person(name) WHERE name = $autostring_1
| |           |           |           |           |           |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
+-----+
Total database accesses: 0, total allocated memory: 184
```

Asserting Multi Node Index Seek

The `AssertingMultiNodeIndexSeek` operator is used to ensure that no property uniqueness 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 property uniqueness constraints on `:Team(name)` and `:Team(id)`, any node that would be found by the `UniqueIndexSeek` operator must be the very same node or the constraints would be violated.

Example 436. AssertingMultiNodeIndexSeek

Query

```
PROFILE  
MERGE (t:Team {name: 'Engineering', id: 42})
```

Query Plan

```
Planner COST  
Runtime PIPELINED  
Runtime version 5.4  
Batch size 128  
+-----  
+-----  
+-----+-----+-----+-----+  
+-----+  
| Operator | Details  
| Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline  
|  
+-----  
+-----+-----+-----+-----+  
+-----+  
| +ProduceResults |  
| 1 | 0 | 0 |  
|  
|  
+-----+-----+-----+  
+-----+-----+-----+  
| +EmptyResult |  
| 1 | 0 | 0 |  
|  
|  
+-----+-----+-----+  
+-----+-----+-----+  
| +Merge | CREATE (t:Team {name: $autostring_0, id: $autoint_1})  
| 1 | 1 | 0 |  
|  
|  
+-----+-----+-----+  
+-----+-----+-----+  
| +AssertingMultiNodeIndexSeek | UNIQUE t:Team(name) WHERE name = $autostring_0, UNIQUE t:Team(id)  
WHERE id = $autoint_1 | 0 | 2 | 4 | 120 | 0/2 |  
1.584 | Fused in Pipeline 0 |  
+-----+-----+-----+  
+-----+-----+-----+  
+-----+  
Total database accesses: 4, total allocated memory: 184
```

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 437. NodeIndexSeekByRange

Query

```
PROFILE
MATCH (l:Location)
WHERE l.name STARTS WITH 'Lon'
RETURN l
```

Query Plan

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

+-----+
+-----+-----+-----+
| Operator          | Details
| Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline
| +-----+-----+-----+-----+-----+-----+
| +ProduceResults | 1
2 |   1 |   0 |       |           |           |           |
| |   +-----+-----+-----+-----+-----+-----+
| | +NodeIndexSeekByRange | RANGE INDEX l:Location(name) WHERE name STARTS WITH $autostring_0 |
2 |   1 |   2 |       120 |           3/0 |     0.825 | Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
Total database accesses: 2, total allocated memory: 184
```

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 438. NodeUniqueIndexSeekByRange

Query

```
PROFILE
MATCH (t:Team)
WHERE t.name STARTS WITH 'Ma'
RETURN t
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator Details Estimated						
Rows		DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline
2	0	0	t			
1			+			
2	0	1	120	1/0	0.623	Fused in Pipeline 0

Total database accesses: 1, total allocated memory: 184

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 439. NodeIndexContainsScan

Query

```
PROFILE
MATCH (l:Location)
WHERE l.name CONTAINS 'al'
RETURN l
```

Query Plan

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

+-----+
+-----+-----+-----+
| Operator      | Details          | Estimated   |
Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline   |
+-----+-----+-----+-----+
+-----+-----+-----+
| +ProduceResults | 1               |           |
0 | 2 | 0 |           |           |           |           |
| | +-----+-----+-----+
| | +NodeIndexContainsScan | TEXT INDEX l:Location(name) WHERE name CONTAINS $autostring_0 |
0 | 2 | 3 |           | 120 | 2/0 | 1.305 | Fused in Pipeline 0 |
+-----+-----+-----+
+-----+-----+-----+
Total database accesses: 3, total allocated memory: 184
```

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 440. NodeIndexEndsWithScan

Query

```
PROFILE
MATCH (l:Location)
WHERE l.name ENDS WITH 'al'
RETURN l
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator Details Estimated						
Rows		DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline
0		0	0			
+NodeIndexEndsWithScan	TEXT INDEX l:Location(name) WHERE name ENDS WITH \$autostring_0	1	120	0/0	4.409	Fused in Pipeline 0

Total database accesses: 1, total allocated memory: 184

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 441. NodeIndexScan

Query

```
PROFILE
MATCH (l:Location)
WHERE l.name IS NOT NULL
RETURN l
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline	Estimated Rows	Rows	DB
+ProduceResults	1	0					10	10	
+NodeIndexScan	RANGE INDEX l:Location(name) WHERE name IS NOT NULL	11	120	2/1	0.557	Fused in Pipeline 0	10	10	

Total database accesses: 11, total allocated memory: 184

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 442. Apply

Query

```
PROFILE
MATCH (p:Person {name: 'me'})
MATCH (q:Person {name: p.secondName})
RETURN p, q
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline	Estimated Rows	Rows
+ProduceResults	p, q	0					1	0
+Apply		0					1	0
+NodeIndexSeek	RANGE INDEX q:Person(name) WHERE name = p.secondName	2152	2152	0/0	0.219	Fused in Pipeline 1	1	0
+NodeIndexSeek	RANGE INDEX p:Person(name) WHERE name = \$autostring_0	2	120	0/1	0.236	In Pipeline 0	1	1

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 443. SemiApply

Query

```
PROFILE
CYPER runtime=slotted
MATCH (p:Person)
WHERE (p)-[:FRIENDS_WITH]->(:Person)
RETURN p.name
```

Query Plan

```
Planner COST
Runtime SLOTTED
Runtime version 5.4

+-----+-----+-----+
| Operator      | Details          | Estimated Rows | Rows | DB Hits | Page
Cache Hits/Misses |
+-----+-----+-----+
+-----+
| +ProduceResults | `p.name`           | 11 | 10 | 0 |
0/0 |
| |
+-----+
| +Projection    | p.name AS `p.name` | 11 | 10 | 10 |
1/0 |
| |
+-----+
| +SemiApply     |                   | 11 | 10 | 0 |
0/0 |
| \\
+-----+
| | +Filter      | anon_3:Person       | 12 | 0 | 10 |
0/0 |
| |
+-----+
| | +Expand(All) | (p)-[anon_2:FRIENDS_WITH]->(anon_3) | 12 | 10 | 51 |
28/0 |
| |
+-----+
| | +Argument    | p                   | 14 | 14 | 0 |
0/0 |
| |
+-----+
| +NodeByLabelScan| p:Person           | 14 | 14 | 35 |
1/0 |
+-----+
+-----+
Total database accesses: 142, 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 444. AntiSemiApply

Query

```
PROFILE
CYpher runtime=slotted
MATCH
  (me:Person {name: 'me'}),
  (other:Person)
WHERE NOT (me)-[:FRIENDS_WITH]->(other)
RETURN other.name
```

Query Plan

Planner COST

Runtime SLOTTED

Runtime version 5.4

Operator	Details	Estimated Rows	Rows
DB Hits			
Memory (Bytes)			
Page Cache Hits/Misses			
+ProduceResults	'other.name'	4	12
0	0/0		
+Projection	other.name AS `other.name`	4	12
12	1/0		
+AntiSemiApply		4	12
0	0/0		
+Expand(Into)	(me)-[anon_2:FRIENDS_WITH]->(other)	1	0
81	896 28/0		
+Argument	me, other	14	14
0	0/0		
+CartesianProduct		14	14
0	0/0		
+NodeByLabelScan	other:Person	14	14
35	1/0		
+NodeIndexSeek	RANGE INDEX me:Person(name) WHERE name = \$autostring_0	1	1
2	0/1		

Total database accesses: 166, total allocated memory: 976

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 445. Anti

Query

```
PROFILE
CYpher runtime=pipelined
MATCH
  (me:Person {name: 'me'}),
  (other:Person)
WHERE NOT (me)-[:FRIENDS_WITH]->(other)
RETURN other.name
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline	Estimated Rows	Rows	
+ProduceResults	'other.name'	0		0/0	0.068		4	12	
+Projection	other.name AS 'other.name'	24		2/0	0.111	In Pipeline 4	4	12	
+Apply		0		0/0			4	12	
+Anti		0	1256		0/0	0.084	In Pipeline 4	4	12
+Limit	1	0	752				11	2	
+Expand(Into)	(me)-[anon_2:FRIENDS_WITH]->(other)	81	2632				1	2	
+Argument	me, other	0	3192	1/0	0.904	Fused in Pipeline 3	14	14	
+CartesianProduct		0	3672		1.466	In Pipeline 2	14	14	
+NodeByLabelScan	other:Person	35					14	14	
+NodeIndexSeek	RANGE INDEX me:Person(name) WHERE name = \$autostring_0	2	120	0/1	0.493	In Pipeline 0	1	1	

Total database accesses: 178, total allocated memory: 6744

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 446. LetSemiApply

Query

```
PROFILE
CYpher runtime=slotted
MATCH (other:Person)
WHERE (other)-[:FRIENDS_WITH]->(:Person) OR (other)-[:WORKS_IN]->(:Location)
RETURN other.name
```

Query Plan

Planner COST

Runtime SLOTTED

Runtime version 5.4

Operator	Details	Estimated Rows	Rows	DB Hits
Page Cache Hits/Misses				
+ProduceResults	`other.name`	13	14	0
0/0				
+Projection	other.name AS `other.name`	13	14	14
1/0				
+SelectOrSemiApply	anon_9	14	14	0
0/0	\			
+Filter	anon_7:Location	14	0	4
0/0				
+Expand(All)	(other)-[anon_6:WORKS_IN]->(anon_7)	14	4	15
8/0	+			
+Argument	other	14	4	0
0/0	+			
+LetSemiApply		14	14	0
0/0	\			
+Filter	anon_5:Person	12	0	10
0/0	+			
+Expand(All)	(other)-[anon_4:FRIENDS_WITH]->(anon_5)	12	10	51
28/0	+			
+Argument	other	14	14	0
0/0	+			
+NodeByLabelScan	other:Person	14	14	35
1/0	+			

Total database accesses: 165, 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 447. LetAntiSemiApply

Query

```
PROFILE
CYpher runtime=slotted
MATCH (other:Person)
WHERE NOT ((other)-[:FRIENDS_WITH]->(:Person)) OR (other)-[:WORKS_IN]->(:Location)
RETURN other.name
```

Query Plan

Planner COST

Runtime SLOTTED

Runtime version 5.4

Operator	Details	Estimated Rows	Rows	DB Hits
Page Cache Hits/Misses				
+ProduceResults `other.name`		11	14	0
0/0				
+Projection other.name AS `other.name`		11	14	14
1/0				
+SelectOrSemiApply anon_9		14	14	0
0/0				
\ +Filter anon_7:Location		14	0	10
0/0				
+Expand(All) (other)-[anon_6:WORKS_IN]->(anon_7)		14	10	38
20/0				
+Argument other		14	10	0
0/0				
+LetAntiSemiApply		14	14	0
0/0				
\ +Filter anon_5:Person		12	0	10
0/0				
+Expand(All) (other)-[anon_4:FRIENDS_WITH]->(anon_5)		12	10	51
28/0				
+Argument other		14	14	0
0/0				
+NodeByLabelScan p:Person		14	14	35
1/0				

Total database accesses: 142, 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 448. SelectOrSemiApply

Query

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

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline	Estimated Rows	Rows	DB Hits
+ProduceResults	'other.name'					11	10	0
+Projection	other.name AS 'other.name'					11	10	20
+SelectOrSemiApply	other.age > \$autoint_0 0/0 0.190 Fused in Pipeline 2	392				14	10	0
+Limit	1	752				14	10	0
+Filter	anon_3:Person					12	10	20
+Expand(All)	(other)-[anon_2:FRIENDS_WITH]->(anon_3)					12	10	37
+Argument	other	2168	2/0 0.435 Fused in Pipeline 1			14	14	0
+NodeByLabelScan	other:Person					14	14	35

Total database accesses: 148, total allocated memory: 2952

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 449. SelectOrAntiSemiApply

Query

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

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows	Rows	DB Hits
Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline	
+ProduceResults	'other.name'			4 4 0
+Projection	other.name AS 'other.name'			4 4 8
+SelectOrAntiSemiApply	other.age > \$autoint_0 200 0/0 0.155 Fused in Pipeline 3	14	4 4 0	
\				
+Anti	1256 0/0 0.170 In Pipeline 2	14	4 4 0	
+Limit	752 1 0 10 0	0	10 0 0	
+Filter	anon_3:Person 12 10 20			
+Expand(All)	(other)-[anon_2:FRIENDS_WITH]->(anon_3) 12 10 37			
+Argument	other 14 14 0			
	2168 2/0 0.449 Fused in Pipeline 1			
+NodeByLabelScan	other:Person 14 14 35			
	Fused in Pipeline 0			

Total database accesses: 136, total allocated memory: 4208

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 450. LetSelectOrSemiApply

Query

```
PROFILE
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

Planner COST

Runtime SLOTTED

Runtime version 5.4

Operator	Details	Estimated Rows	Rows	DB Hits
Page Cache Hits/Misses				
+ProduceResults	`other.name`	13	14	0
0/0				
+Projection	other.name AS `other.name`	13	14	14
1/0				
+SelectOrSemiApply	anon_9	14	14	0
0/0				
+Filter	anon_7:Location	14	0	4
0/0				
+Expand(All)	(other)-[anon_6:WORKS_IN]->(anon_7)	14	4	15
8/0				
+Argument	other	14	4	0
0/0				
+LetSelectOrSemiApply	other.age = \$autoint_0	14	14	14
0/0				
+Filter	anon_5:Person	12	0	10
0/0				
+Expand(All)	(other)-[anon_4:FRIENDS_WITH]->(anon_5)	12	10	51
28/0				
+Argument	other	14	14	0
0/0				
+NodeByLabelScan	other:Person	14	14	35
1/0				

Total database accesses: 179, 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 451. LetSelectOrAntiSemiApply

Query

```
PROFILE
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

Planner COST

Runtime SLOTTED

Runtime version 5.4

Operator		Details	Estimated Rows	Rows	DB
Hits					
		Page Cache Hits/Misses			
0	0/0	`other.name`		12	14
14	1/0	other.name AS `other.name`		12	14
0	0/0	+SelectOrSemiApply anon_9		14	14
10	0/0	+Filter anon_7:Location		14	0
38	20/0	+Expand(All) (other)-[anon_6:WORKS_IN]->(anon_7)		14	10
0	0/0	+Argument other		14	10
14	0/0	+LetSelectOrAntiSemiApply other.age = \$autoint_0		14	14
10	0/0	+Filter anon_5:Person		12	0
51	28/0	+Expand(All) (other)-[anon_4:FRIENDS_WITH]->(anon_5)		12	10
0	0/0	+Argument other		14	14
35	1/0	+NodeByLabelScan other:Person		14	14
Total database accesses: 208, 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 for each incoming row. If no matches are found instead nodes and relationships are created and all `ON CREATE` operations are run.

Example 452. Merge

Query

```
PROFILE
MERGE (p:Person {name: 'Andy'})
ON MATCH SET p.existed = true
ON CREATE SET p.existed = false
```

Query Plan

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

+-----+
+-----+-----+-----+
| Operator | Details
| Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline
| +-----+-----+-----+-----+-----+-----+
| +ProduceResults |
1 | 0 | 0 | | | | |
| +-----+-----+-----+-----+-----+-----+
| +EmptyResult |
1 | 0 | 0 | | | | |
| +-----+-----+-----+-----+-----+-----+
| +Merge |
1 | 1 | 2 | | | |
| | | | ON MATCH SET p.existed = true, | |
| | | | | |
| | | | | ON CREATE SET p.existed = false |
| | | | | |
| +-----+-----+-----+-----+-----+-----+
| +NodeIndexSeek | RANGE INDEX p:Person(name) WHERE name = $autostring_0 |
1 | 1 | 2 | 120 | 2/1 | 0.749 | Fused in Pipeline 0 |
| +-----+-----+-----+-----+-----+-----+
| +-----+-----+-----+-----+-----+-----+
Total database accesses: 4, total allocated memory: 184
```

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 453. LockingMerge

Query

```
PROFILE
MATCH (s:Person {name: 'me'})
MERGE (s)-[:FRIENDS_WITH]->(s)
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline	Estimated Rows	Rows
+ProduceResults		0					1	0
+EmptyResult		0					1	0
+Apply		0					1	1
+LockingMerge	CREATE (s)-[anon_0:FRIENDS_WITH]->(s), LOCK(s)	1					1	1
+Expand(Into)	(s)-[anon_0:FRIENDS_WITH]->(s)	12	896				0	0
+Argument	s	0	2152	3/0	0.616	Fused in Pipeline 1	1	3
+NodeIndexSeek	RANGE INDEX s:Person(name) WHERE name = \$autostring_0	2	120	0/1	0.298	In Pipeline 0	1	1

Total database accesses: 15, 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 454. RollUpApply

Query

```
PROFILE
CYpher runtime=slotted
MATCH (p:Person)
RETURN p.name, [(p)-[:WORKS_IN]->(location) | location.name] AS cities
```

Query Plan

Planner COST

Runtime SLOTTED

Runtime version 5.4

Operator	Details	Estimated Rows	Rows	DB Hits	Page Cache
Hits/Misses					
+ProduceResults	`p.name` , cities	14	14	0	
0/0					
+Projection	p.name AS `p.name`	14	14	14	
0/0					
+RollUpApply	cities, anon_0	14	14	0	
0/0					
+Projection	location.name AS anon_0	15	15	15	
1/0					
+Expand(All)	(p)-[anon_2:WORKS_IN]->(location)	15	15	53	
28/0					
+Argument	p	14	14	0	
0/0					
+NodeByLabelScan	p:Person	14	14	35	
1/0					

Total database accesses: 153, 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 455. Argument

Query

```
PROFILE
MATCH (s:Person {name: 'me'})
MERGE (s)-[:FRIENDS_WITH]->(s)
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline	Estimated Rows	Rows
+ProduceResults		0					1	0
+EmptyResult		0					1	0
+Apply		0					1	1
+LockingMerge	CREATE (s)-[anon_0:FRIENDS_WITH]->(s), LOCK(s)	1					1	1
+Expand(Into)	(s)-[anon_0:FRIENDS_WITH]->(s)	12	896				0	0
+Argument	s	0	2152	3/0	0.631	Fused in Pipeline 1	1	3
+NodeIndexSeek	RANGE INDEX s:Person(name) WHERE name = \$autostring_0	2	120	0/1	0.220	In Pipeline 0	1	1

Total database accesses: 15, 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 456. Expand(All)

Query

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

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline	Estimated Rows	Rows
+ProduceResults	fof	0					1	2
+Expand(All)	(p)-[:FRIENDS_WITH]->(fof)	5					1	2
+NodeIndexSeek	RANGE INDEX p:Person(name) WHERE name = \$autostring_0	2	120	4/1	1.137	Fused in Pipeline 0	1	1

Total database accesses: 7, total allocated memory: 184

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 457. Expand(Into)

Query

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

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows Rows					
		DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline	
+ProduceResults	fof						0 0
0							
+Filter	not anon_1 = anon_0						0 0
0							
+Expand(Into)	(p)-[anon_0:FRIENDS_WITH]->(fof)						0 0
6	896						
+Expand(All)	(p)<-[anon_1]-(fof)						1 1
5							
+NodeIndexSeek	RANGE INDEX p:Person(name) WHERE name = \$autostring_0						1 1
2	120	4/1	0.546	Fused in Pipeline	0		

Total database accesses: 13, total allocated memory: 976

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 458. OptionalExpand(All)

Query

```
PROFILE
MATCH (p:Person)
OPTIONAL MATCH (p)-[works_in:WORKS_IN]->(1)
  WHERE works_in.duration > 180
RETURN p, 1
```

Query Plan

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

+-----+
+-----+-----+-----+
| Operator      | Details
Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline
|              |
+-----+-----+-----+-----+
+-----+
| +ProduceResults | p, 1
14 |   15 |     1 |           |           |           |           |
| |           +-----+
+-----+-----+-----+-----+
| +OptionalExpand(All) | (p)-[works_in:WORKS_IN]->(1) WHERE works_in.duration > $autoint_0 |
14 |   15 |    53 |           |           |           |           |
| |           +-----+
+-----+-----+-----+-----+
| +NodeByLabelScan | p:Person
14 |   14 |    15 |       120 |           5/0 |     1,233 | Fused in Pipeline 0 |
+-----+-----+-----+-----+
+-----+
Total database accesses: 125, total allocated memory: 184
```

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 459. OptionalExpand(Into)

Query

```
PROFILE
MATCH (p:Person)-[works_in:WORKS_IN]->(l)
OPTIONAL MATCH (l)-->(p)
RETURN p
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline
+ProduceResults	p			15	15	0		
+OptionalExpand(Into)	(1)-[anon_0]->(p)			15	15	105		
3360								
+Expand(All)	(p)-[works_in:WORKS_IN]->(l)			15	15	39		
+NodeByLabelScan	p:Person			14	14	15		
120	7/0	3,925	Fused in Pipeline 0					

Total database accesses: 215, total allocated memory: 3440

VarLength Expand All

Given a start node, the `VarLengthExpand(All)` operator will traverse variable-length relationships.

Example 460. VarLengthExpand(All)

Query

```
PROFILE
MATCH (p:Person)-[:FRIENDS_WITH *1..2]-(q:Person)
RETURN p, q
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows	Rows	DB Hits
Memory (Bytes)	Page Cache Hits/Misses			
+ProduceResults	p, q	40	48	0
+Filter	q:Person	40	48	96
+VarLengthExpand(All)	(p)-[anon_0:FRIENDS_WITH*..2]-(q)	40	48	151
128				
+NodeByLabelScan	p:Person	14	14	15
120	6/0 10,457 Fused in Pipeline 0			

Total database accesses: 318, 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 461. VarLengthExpand(Into)

Query

```
PROFILE
MATCH (p:Person)-[:FRIENDS_WITH *1..2]-(p:Person)
RETURN p
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows	Rows	DB Hits
Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline	
+ProduceResults	p		3	4
				0
+VarLengthExpand(Into)	(p)-[anon_0:FRIENDS_WITH*..2]-(p)		3	4
128				151
+NodeByLabelScan	p:Person		14	14
120	6/0 0,797 Fused in Pipeline 0		15	

Total database accesses: 222, 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 kind of expand is only planned when:

- The individual paths are not of interest.
- The relationships have an upper bound.

The [VarLengthExpand\(Pruning\)](#) operator guarantees that all the end nodes produced will be unique.

Example 462. VarLengthExpand(Pruning)

Query

```
PROFILE
MATCH (p:Person)-[:FRIENDS_WITH *3..4]-(q:Person)
RETURN DISTINCT p, q
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator (Bytes)	Details	Estimated Rows	Rows	DB Hits	Memory
Page Cache Hits/Misses	Time (ms)				
+ProduceResults	p, q 2/0 0.242	17	32	0	
+Distinct	p, q 2208 0/0 13.409	17	32	0	
+Filter	q:Person 1/0 0.358	18	32	64	
+VarLengthExpand(Pruning)	(p)-[:FRIENDS_WITH*3..4]-(q) 1696 In Pipeline 1	18	32	204	
+NodeByLabelScan	p:Person 120 1/0 0,100 In Pipeline 0	14	14	15	

Total database accesses: 339, total allocated memory: 2288

Breadth First VarLength Expand Pruning

Given a start node, the **VarLengthExpand(Pruning,BFS)** operator traverses variable-length relationships much like the **VarLengthExpand(All)** operator. However, as an optimization, it instead performs a breadth-first search (BFS) and while expanding, some paths are not explored if they are guaranteed to produce an end node that has already been found (by means of a previous path traversal). This is only used in cases where the individual paths are not of interest.

This kind of expand is only planned when:

- The individual paths are not of interest.
- The relationships have an upper bound.
- The lower bound is either **0** or **1** (default).

This operator guarantees that all the end nodes produced are unique.

Query

```
PROFILE
MATCH (p:Person)-[:FRIENDS_WITH *..4]-(q:Person)
RETURN DISTINCT p, q
```

Query Plan

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

+-----+-----+-----+-----+
| Operator | Details | Estimated Rows | Rows | DB Hits | Memory
| (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
| +ProduceResults | p, q | 56 | 68 | 0 |
| | |
| +-----+-----+-----+-----+
| +Distinct | p, q | 56 | 68 | 0 |
4480 |
| | |
| +-----+-----+-----+-----+
| +Filter | q:Person | 59 | 68 | 136 |
| | |
| +-----+-----+-----+-----+
| +VarLengthExpand(Pruning,BFS) | (p)-[:FRIENDS_WITH*..4]-(q) | 59 | 68 | 280 |
696 |
| | |
| +-----+-----+-----+-----+
| +NodeByLabelScan | p:Person | 14 | 14 | 15 |
120 |
4/0 | 3,202 | Fused in Pipeline 0 |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
Total database accesses: 487, total allocated memory: 5256
```

Assert Same Node

The `AssertSameNode` operator is used to ensure that no property uniqueness 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 property uniqueness constraints on `:Team(name)` and `:Team(id)`, any node that would be found by the `UniqueIndexSeek` operator must be the very same node or the constraints would be violated.

Example 463. AssertSameNode

Query

```
PROFILE
CYpher runtime=slotted
MERGE (t:Team {name: 'Engineering', id: 42})
```

Query Plan

Planner COST

Runtime SLOTTED

Runtime version 5.4

Operator	Details	Estimated
Rows	Rows	DB Hits
		Page Cache Hits/Misses
+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 464. EmptyResult

Query

```
PROFILE  
CREATE (:Person)
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator (ms)	Details Pipeline	Estimated Rows	Rows	DB Hits	Page Cache Hits/Misses	Time
+ProduceResults		1	0	0		
+EmptyResult		1	0	0		
+Create	(anon_0:Person)	1	1	1		0/0
0.000	Fused in Pipeline 0					

Total database accesses: 1, total allocated memory: 184

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 465. ProduceResults

Query

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

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline
+ProduceResults	n	35	35	0				
+AllNodesScan	n	35	35	36	120			Fused in Pipeline 0

Total database accesses: 36, total allocated memory: 184

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 466. LoadCSV

Query

```
PROFILE
LOAD CSV FROM 'https://neo4j.com/docs/cypher-refcard/3.3/csv/artists.csv' AS line
RETURN line
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline
+ProduceResults	line	10	4	0				
0/0	0.210							
+LoadCSV	line	10	4	0	72			
	In Pipeline 1							

Total database accesses: 0, total allocated memory: 184

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 467. NodeHashJoin

Query

```
PROFILE
MATCH (bob:Person {name: 'Bob'})-[:WORKS_IN]->(loc)<-[>:WORKS_IN]-(matt:Person {name: 'Mattias'})
RETURN loc.name
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline	Estimated Rows	Rows
+ProduceResults	'loc.name'	0			0/0	0.000	10	0
+Projection	loc.name AS 'loc.name'	0			0/0	0.000	10	0
+Filter	not anon_0 = anon_1	0			0/0	0.000	10	0
+NodeHashJoin	loc	0	3688		0.053	In Pipeline 2	10	0
+Expand(All)	(matt)-[anon_1:WORKS_IN]->(loc)	0					19	0
+NodeIndexSeek	RANGE INDEX matt:Person(name) WHERE name = \$autostring_1	1					1	0
+Expand(All)	(bob)-[anon_0:WORKS_IN]->(loc)	4					19	1
+NodeIndexSeek	RANGE INDEX bob:Person(name) WHERE name = \$autostring_0	2	120		3/0	0.556	1	1

Total database accesses: 7, total allocated memory: 3888

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 468. ValueHashJoin

Query

```
PROFILE
MATCH
  (p:Person),
  (q:Person)
WHERE p.age = q.age
RETURN p, q
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache
Hits/Misses	Time (ms)	Pipeline				
+ProduceResults	p, q	10	0	0		
0/0	0.000					
+ValueHashJoin	p.age = q.age In Pipeline 2	10	0	0	344	
\						
+NodeByLabelScan	q:Person In Pipeline 1	15	0	0	120	
0/0	0,000					
+NodeByLabelScan	p:Person In Pipeline 0	15	15	16	120	
1/0	0,211					

Total database accesses: 71, total allocated memory: 664

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 469. NodeRightOuterHashJoin

Query

```
PROFILE
MATCH (a:Person)
OPTIONAL MATCH (a)-->(b:Person)
USING JOIN ON a
RETURN a.name, b.name
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator		Details			Estimated Rows	
Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline	
16	0	0	0/0	0.102		14
+ProduceResults						
16	8	0	0/0	0.055		14
+Projection						
16	0	4232	1/0	0.269	In Pipeline 2	14
\						
+NodeRightOuterHashJoin						
15	16	120	5/0	0,049	In Pipeline 1	15
+NodeByLabelScan						
13	39	0	0/0	0.001		13
+CacheProperties						
13	55	0	0/0	0.001		13
+Expand(A11)						
15	16	120	5/0	1,150	Fused in Pipeline 0	15

Total database accesses: 211, total allocated memory: 4312

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 470. TriadicSelection

Query

```
PROFILE
CYpher runtime=slotted
MATCH (me:Person)-[:FRIENDS_WITH]-(o)-[:FRIENDS_WITH]-(other)
WHERE NOT (me)-[:FRIENDS_WITH]-(other)
RETURN other.name
```

Query Plan

Planner COST

Runtime SLOTTED

Runtime version 5.4

Operator	Details	Estimated Rows	Rows	DB Hits	Page
Cache Hits/Misses					
+ProduceResults	`other.name`	4	24	0	
0/0					
+Projection	other.name AS `other.name`	4	24	24	
1/0					
+Filter	not anon_2 = anon_4	16	24	0	
0/0					
+TriadicSelection	WHERE NOT (me)--(other)	4	24	0	
0/0					
+Expand(All)	(anon_3)-[anon_4:FRIENDS_WITH]-(other)	16	48	98	
48/0					
+Argument	anon_3, anon_2	24	24	0	
0/0					
+Expand(All)	(me)-[anon_2:FRIENDS_WITH]-(anon_3)	24	24	53	
28/0					
+NodeByLabelScan	me:Person	15	15	16	
1/0					

Total database accesses: 246, 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 471. TriadicBuild

Query

```
PROFILE
CYpher runtime=pipelined
MATCH (me:Person)-[:FRIENDS_WITH]-(o)-[:FRIENDS_WITH]-(other)
WHERE NOT (me)-[:FRIENDS_WITH]-(other)
RETURN other.name
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows	Rows	DB Hits	Memory
(Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline		
+ProduceResults	`other.name`	4	24	0	
	0/0 0.133				
+Projection	other.name AS `other.name`	4	24	48	
	2/0 0.056				
+Filter	not anon_2 = anon_4	16	24	0	
+TriadicFilter	WHERE NOT (me)--(other)	4	24	0	
4136	0/0 0.195 In Pipeline 3				
+Apply		16	24	0	
	0/0				
+Expand(All)	(anon_3)-[anon_4:FRIENDS_WITH]-(other)	16	48	98	
+Argument	anon_3, anon_2	24	24	0	
4200	0/0 0.397 Fused in Pipeline 2				
+TriadicBuild	(me)--(anon_3)	24	24	0	
888	0/0 1.427 In Pipeline 1				
+Expand(All)	(me)-[anon_2:FRIENDS_WITH]-(anon_3)	24	24	39	
+NodeByLabelScan	me:Person	15	15	16	
120	3/0 0,200 Fused in Pipeline 0				

Total database accesses: 256, total allocated memory: 7376

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 472. TriadicFilter

Query

```
PROFILE
CYpher runtime=pipelined
MATCH (me:Person)-[:FRIENDS_WITH]-(o)-[:FRIENDS_WITH]-(other)
WHERE NOT (me)-[:FRIENDS_WITH]-(other)
RETURN other.name
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows	Rows	DB Hits	Memory
(Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline		
+ProduceResults	`other.name`	4	24	0	
	0/0 0.189				
+Projection	other.name AS `other.name`	4	24	48	
	2/0 0.381				
+Filter	not anon_2 = anon_4	16	24	0	
+TriadicFilter	WHERE NOT (me)--(other)	4	24	0	
4136	0/0 0.685 In Pipeline 3				
+Apply		16	24	0	
	0/0				
+Expand(All)	(anon_3)-[anon_4:FRIENDS_WITH]-(other)	16	48	98	
+Argument	anon_3, anon_2	24	24	0	
4200	0/0 0.496 Fused in Pipeline 2				
+TriadicBuild	(me)--(anon_3)	24	24	0	
888	0/0 3.268 In Pipeline 1				
+Expand(All)	(me)-[anon_2:FRIENDS_WITH]-(anon_3)	24	24	39	
+NodeByLabelScan	me:Person	15	15	16	
120	3/0 0,481 Fused in Pipeline 0				

Total database accesses: 256, total allocated memory: 7376

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 473. CartesianProduct

Query

```
PROFILE
MATCH
  (p:Person),
  (t:Team)
RETURN p, t
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses
						Time (ms)
+ProduceResults	p, t	140	140	0		
2/0	1.917					
+CartesianProduct		140	140	0	1736	
1/2	1.209 In Pipeline 2					
+\\						
+ +NodeByLabelScan	t:Team	10	10	11	136	
1/0	1,145 In Pipeline 1					
+ +						
+ +NodeByLabelScan	p:Person	15	15	16	120	
1/0	0,409 In Pipeline 0					

Total database accesses: 142, total allocated memory: 1816

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 474. Foreach

Query

```
PROFILE
CYpher runtime=slotted
FOREACH (value IN [1,2,3] | CREATE (:Person {age: value}))
```

Query Plan

Planner COST

Runtime SLOTTED

Runtime version 5.4

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], CREATE (anon_0:Person {age: value})	1	1
9	0/0		

Total database accesses: 9, total allocated memory: 64

Eager

The [Eager](#) operator causes all preceding operators to execute fully, for the whole dataset, before continuing execution. This is done to ensure isolation between parts of the query plan that might otherwise affect each other.

Values from the graph are fetched in a lazy manner; i.e. a pattern matching might not be fully exhausted before updates are applied. To maintain correct semantics, the query planner will insert [Eager](#) operators into the query plan to prevent updates from influencing pattern matching, or other read operations. This scenario is exemplified by the query below, where the [DELETE](#) clause would otherwise influence both the [MATCH](#) clause and the [MERGE](#) clause. For more information on how the [Eager](#) operator can ensure correct semantics, see the section on [Clause composition](#).

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 475. Eager

Query

```
PROFILE
MATCH (a), (b)
DELETE a, b
MERGE ()
```

Query Plan

```
Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 1024

+-----+
+-----+
+-----+-----+-----+-----+-----+-----+
| Operator      | Id | Details
| Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline
+-----+
+-----+
+-----+-----+-----+-----+-----+-----+
| +ProduceResults | 0 |
| 15129 | 0 | 0 | | | |
| | +---+
+-----+
+-----+-----+-----+-----+-----+-----+
| +EmptyResult | 1 |
| 15129 | 0 | 0 | | | |
| | +---+
+-----+
+-----+-----+-----+-----+-----+-----+
| +Apply | 2 |
| 15129 | 15129 | 0 | | | |
| | \ +---+
+-----+
+-----+-----+-----+-----+-----+-----+
| | +Merge | 3 | CREATE (anon_0)
| | 15129 | 15129 | 1 | | | |
| | | +---+
+-----+
+-----+-----+-----+-----+-----+-----+
| | | +AllNodesScan | 4 | anon_0
| | | 1860867 | 15128 | 30257 | 24680 | 347967/0 | 1820.510 | Fused in
Pipeline 5 |
| | | +---+
+-----+
+-----+-----+-----+-----+-----+-----+
| | +Eager | 5 | read/delete conflict for variable: anon_0, read/delete conflict for
variable: a, | 15129 | 15129 | 0 | 243608 | 0/0 | 0.182
| | In Pipeline 4 | | | | | | |
| | | | | | | | | | |
| | | +---+
+-----+
+-----+-----+-----+-----+-----+-----+
| | +Delete | 6 | b
| | 15129 | 15129 | 122 | | | |
| | | +---+
+-----+
+-----+-----+-----+-----+-----+-----+
| | +Delete | 7 | a
```

	15129	15129	1				
+----							
+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+
+Eager 8 read/delete conflict for variable: b							
15129	15129	0	243608		0/0	8.515	Fused in
Pipeline 3 +----							
+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+
+CartesianProduct 9							
15129	15129	0	10712		0/0	0.342	In
Pipeline 2 \ +----							
+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+
+AllNodesScan 10 b							
123	123	124	136		23/0	0.243	In
Pipeline 1 +----							
+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+
+AllNodesScan 11 a							
123	123	124	120		23/0	0.344	In
Pipeline 0 +----							
+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+-----+

Total database accesses: 30629, total allocated memory: 260488

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 476. EagerAggregation

Query

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

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows Rows				
		DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline
+ProduceResults	location, people	0	0/0	0.113	In Pipeline 1	4 6
+EagerAggregation	cache[l.name] AS location, collect(p.name) AS people	30	2584			4 6
+Filter	p:Person	30				15 15
+Expand(All)	(l)-[:WORKS_IN]-(p)	16				15 15
+CacheProperties	cache[l.name]	10				10 10
+NodeByLabelScan	l:Location	11	120	5/0	3,077 Fused in Pipeline 0	10 10

Total database accesses: 157, total allocated memory: 2664

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 477. OrderedAggregation

Query

```
PROFILE
MATCH (p:Person)
WHERE p.name STARTS WITH 'P'
RETURN p.name, count(*) AS count
```

Query Plan

```
Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

+-----+
+-----+-----+
+-----+-----+-----+
| Operator | Details
| Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Ordered by
| Pipeline | |
+-----+
+-----+-----+-----+
+-----+
| +ProduceResults | `p.name`, count
| 0 | 2 | 0 | | 0/0 | 0.045 |
| | | | | |
+-----+
+-----+-----+-----+
+-----+
| +OrderedAggregation | cache[p.name] AS `p.name`, count(*) AS count
| 0 | 2 | 0 | 288 | 0/0 | 0.175 | `p.name` ASC | In Pipeline 1 |
| | | | | |
+-----+
+-----+-----+-----+
+-----+
| +NodeIndexSeekByRange | RANGE INDEX p:Person(name) WHERE name STARTS WITH $autostring_0,
cache[p.name] | 0 | 2 | 3 | 120 | 0/1 | 0.529
| p.name ASC | In Pipeline 0 |
+-----+
+-----+-----+-----+
+-----+
```

Total database accesses: 3, total allocated memory: 352

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 478. NodeCountFromCountStore

Query

```
PROFILE
MATCH (p:Person)
RETURN count(p) AS people
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator (Bytes)	Details			Estimated Rows	Rows	DB Hits	Memory
	Page Cache Hits/Misses	Time (ms)	Pipeline				
+ProduceResults	people			1	1	0	
+NodeCountFromCountStore	count((:Person)) AS people	0/0	0.169 Fused in Pipeline 0	1	1	1	

Total database accesses: 1, total allocated memory: 184

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 479. RelationshipCountFromCountStore

Query

```
PROFILE  
MATCH (p:Person)-[r:WORKS_IN]->()  
RETURN count(r) AS jobs
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Rows	DB Hits	Memory (Bytes)	Details		Pipeline	Estimated Rows
			Page Cache Hits/Misses	Time (ms)		
1	0					1
1	1	120	0/0	0.625	Fused in Pipeline 0	1

Total database accesses: 1, total allocated memory: 184

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 480. Distinct

Query

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

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows	Rows	DB Hits	Memory (Bytes)
Page Cache Hits/Misses					
+ProduceResults	1	14	6	0	
+Distinct	1	14	6	0	224
+Filter	p:Person	15	15	30	
+Expand(All)	(l)-[:WORKS_IN]->(p)	15	15	16	
+NodeByLabelScan	l:Location	10	10	11	120
4/0 0,744 Fused in Pipeline 0					

Total database accesses: 117, 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 481. OrderedDistinct

Query

```
PROFILE
MATCH (p:Person)
WHERE p.name STARTS WITH 'P'
RETURN DISTINCT p.name
```

Query Plan

```
Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

+-----+
+-----+
+-----+-----+-----+-----+
| Operator          | Details
| Estimated Rows   | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Ordered by
| Pipeline          |
+-----+
+-----+-----+-----+-----+
+-----+
| +ProduceResults  | `p.name`           |           |           | 0/0 | 0.046 | |
|     0 | 2 | 0 |           |           |           |           |
|           |           |           |           |           |           |
|           |           |           |           |           |           |
+-----+
+-----+-----+-----+-----+
+-----+
| +OrderedDistinct | cache[p.name] AS `p.name`           |           |           | 0/0 | 0.090 | `p.name`
|     0 | 2 | 0 |           |           |           |           |
|           |           |           |           |           |           |
|           |           |           |           |           |           |
| ASC |           |           |           |           |           |
|           |           |           |           |           |           |
+-----+
+-----+-----+-----+-----+
| +NodeIndexSeekByRange | RANGE INDEX p:Person(name) WHERE name STARTS WITH $autostring_0,
cache[p.name] |           0 | 2 | 3 |           120 |           0/1 | 0.493
| p.name ASC | In Pipeline 0 |
+-----+
+-----+-----+-----+-----+
+-----+
```

Total database accesses: 3, total allocated memory: 184

Filter

The **Filter** operator filters each row coming from the child operator, only passing through rows that evaluate the predicates to **true**.

Example 482. Filter

Query

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

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details					Estimated Rows
Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline	
+ProduceResults p	0	0				14
+Filter cache[p.name] =~ \$autostring_0	0	0				14
+NodeIndexScan RANGE INDEX p:Person(name) WHERE name IS NOT NULL, cache[p.name]	14	15	120	0/1	0.763 Fused in Pipeline 0	14

Total database accesses: 15, total allocated memory: 184

Limit

The **Limit** operator returns the first **n** rows from the incoming input.

Example 483. Limit

Query

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

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache
Hits/Misses	Time (ms)	Pipeline				
+ProduceResults	p	3	3	0		
+Limit	3	3	3	0	32	
+NodeByLabelScan	p:Person	3	4	5	120	
3/0	0,540	Fused in Pipeline 0				

Total database accesses: 8, total allocated memory: 184

Skip

The **Skip** operator skips **n** rows from the incoming rows.

Example 484. Skip

Query

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

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Ordered by	Pipeline
+ProduceResults	p	13	13	0					
2/0	0.165								
	+-----+								
+Skip	\$autoint_0	13	13	0		32			
0/0	0.043								
	+-----+								
+Sort	`p.id` ASC	14	14	0		400			
0/0	0.155 p.id ASC	In Pipeline 1							
	+-----+								
+Projection	p.id AS `p.id`	14	14	0					
	+-----+								
+NodeByLabelScan	p:Person	18	18	19		120			
3/0	0,157	Fused in Pipeline 0							

Total database accesses: 71, total allocated memory: 512

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 485. Sort

Query

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

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Ordered by	Pipeline
+ProduceResults	s p		14	14	0				
2/0	0.178								
	+-----+								
+Sort	`p.name` ASC		14	14	0		1192		
0/0	0.107 p.name ASC	In Pipeline 1							
	+-----+								
+Projection	p.name AS `p.name`		14	14	14				
	+-----+								
+NodeByLabelScan	p:Person		14	14	35		120		
3/0	0.221	Fused in Pipeline 0							
+-----+									

Total database accesses: 85, total allocated memory: 1272

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 486. PartialSort

Query

```
PROFILE
MATCH (p:Person)
WHERE p.name STARTS WITH 'P'
RETURN p
ORDER BY p.name, p.age
```

Query Plan

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

+-----+
+-----+-----+
+-----+-----+
| Operator      | Details
| Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Ordered by
| Pipeline      |       |
+-----+
+-----+-----+-----+
+-----+-----+
| +ProduceResults | p
|   0 | 2 | 0 |           |           2/0 | 0.087 |
|   |   |   |           |           |
+-----+
+-----+-----+-----+
| +PartialSort   | `p.name` ASC, `p.age` ASC
|   0 | 2 | 0 |           544 |           0/0 | 0.184 | 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           |   |   |   |           |           |
|   |   |   |           |
+-----+
+-----+-----+-----+
| +NodeIndexSeekByRange | RANGE INDEX p:Person(name) WHERE name STARTS WITH $autostring_0,
| cache[p.name] |           0 | 2 | 3 |           120 |           0/1 | 0.362
| p.name ASC     | Fused in Pipeline 0 |           |
+-----+
+-----+-----+-----+
+-----+-----+
+-----+-----+
```

Total database accesses: 3, total allocated memory: 608

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 487. Top

Query

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

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Ordered by	Pipeline
+ProduceResults	p	2	2	0					
2/0 0.093									
+Top	`p.name` ASC LIMIT 2	2	2	0	1184		0.295	p.name ASC	In Pipeline 1
0/0 0.295									
+Projection	p.name AS `p.name`	14	14	14					
+NodeByLabelScan	p:Person	14	14	35	120		0,166		Fused in Pipeline 0
3/0 0,166									

Total database accesses: 85, total allocated memory: 1264

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 488. PartialTop

Query

```
PROFILE
MATCH (p:Person)
WHERE p.name STARTS WITH 'P'
RETURN p
ORDER BY p.name, p.age
LIMIT 2
```

Query Plan

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

+-----+
+-----+
+-----+-----+-----+-----+
+-----+
| Operator      | Details
| Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Ordered by
| Pipeline      |           |
+-----+
+-----+
+-----+-----+-----+-----+
+-----+
| +ProduceResults   | p
|   0 |   2 |     0 |           |           | 2/0 |    0.093 |
|   |   |     |           |           |           |           |
|   |
+-----+
+-----+-----+-----+-----+
+-----+
| +PartialTop       | `p.name` ASC, `p.age` ASC LIMIT 2
|   0 |   2 |     0 |           | 640 |           | 0/0 |    0.870 | p.name ASC,
| p.age ASC | In Pipeline 1 |           |           |
|   |
+-----+
+-----+-----+-----+-----+
+-----+
| +Projection      | cache[p.name] AS `p.name`, p.age AS `p.age`
|   0 |   2 |     0 |           |           |           |           |
|   |   |     |           |           |           |           |
|   |
+-----+
+-----+-----+-----+-----+
+-----+
| +NodeIndexSeekByRange | RANGE INDEX p:Person(name) WHERE name STARTS WITH $autostring_0,
| cache[p.name] |           0 |   2 |     3 |           120 |           0/1 |    0.556
| p.name ASC      |           |   |     |           |           |           |
|   |
+-----+
+-----+-----+-----+-----+
+-----+
Total database accesses: 3, total allocated memory: 704
```

Union

The **Union** operator concatenates the results from the right child operator with the results from the left child operator.

Example 489. Union

Query

```
PROFILE
MATCH (p:Location)
  RETURN p.name
UNION ALL
MATCH (p:Country)
  RETURN p.name
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline
+ProduceResults	'p.name'	20	11	0				
+Union		20	11	0	776		0.171	Fused in Pipeline 2
+Projection	'p.name'	10	1	0				
+Projection	p.name AS 'p.name'	10	1	1				
+Filter	p:Country	10	1	35				
+AllNodesScan	p	35	35	36	120		0.127	Fused in Pipeline 1
+Projection	'p.name'	10	10	0				
+Projection	p.name AS 'p.name'	10	10	10				
+NodeByLabelScan	p:Location	10	10	11	120		0.171	Fused in Pipeline 0

Total database accesses: 153, total allocated memory: 984

Unwind

The `Unwind` operator returns one row per item in a list.

Example 490. Unwind

Query

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

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Cache Hits/Misses	Time (ms)	Pipeline	Estimated Rows	Rows	DB Hits	Page
+ProduceResults	value				10	5	0	0
+Unwind	range(\$autoint_0, \$autoint_1) AS value				10	5	0	0
0/0	0.000	Fused in Pipeline 0						

Total database accesses: 0, total allocated memory: 184

Exhaustive Limit

The `ExhaustiveLimit` operator is just like a normal `Limit` but will always exhaust the input. Used when combining `LIMIT` and updates

Example 491. ExhaustiveLimit

Query

```
PROFILE
MATCH (p:Person)
SET p.seen = true
RETURN p
LIMIT 3
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache
Hits/Misses	Time (ms)	Pipeline				
+ProduceResults	p	3	3	0		
+ExhaustiveLimit	3	3	3	0	32	
+SetProperty	p.seen = true	14	14	28		
+NodeByLabelScan	p:Person	14	14	35	120	
5/0	41,656	Fused in Pipeline 0				

Total database accesses: 99, total allocated memory: 200

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 492. Optional

Query

```
PROFILE
MATCH (p:Person {name: 'me'})
OPTIONAL MATCH (q:Person {name: 'Lulu'})
RETURN p, q
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline	Estimated Rows	Rows
+ProduceResults	p, q	0	2/0	0.079	In Pipeline 2		1	1
+Apply		0	0/0	0.096			1	1
+Optional	p	0	768	0/0	0.043	In Pipeline 2	1	1
+NodeIndexSeek	RANGE INDEX q:Person(name) WHERE name = \$autostring_1	1	2152	1/0	0.098	In Pipeline 1	1	0
+NodeIndexSeek	RANGE INDEX p:Person(name) WHERE name = \$autostring_0	2	120	0/1	0.364	In Pipeline 0	1	1

Total database accesses: 3, total allocated memory: 3000

Project Endpoints

The **ProjectEndpoints** operator projects the start and end node of a relationship.

Example 493. ProjectEndpoints

Query

```
PROFILE
CREATE (n)-[p:KNOWS]->(m)
WITH p AS r
MATCH (u)-[r]->(v)
RETURN u, v
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator (Bytes)	Details	Estimated Rows	Rows	DB Hits	Memory
	Page Cache Hits/Misses	Time (ms)			
+ProduceResults	u, v		1	1	0
+Apply			1	1	0
+ProjectEndpoints	(u)-[r]->(v)		1	1	0
+Argument	r 4200 0/0 0.248 Fused in Pipeline 1		1	1	0
+Projection	p AS r		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 494. Projection

Query

```
PROFILE  
RETURN 'hello' AS greeting
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows	Rows	DB Hits	Page Cache
Hits/Misses	Time (ms)	Pipeline			
+ProduceResults	greeting		1	1	0
+Projection	\$autostring_0 AS greeting		1	1	0
0/0	0.000	Fused in Pipeline 0			

Total database accesses: 0, total allocated memory: 184

Shortest path

The **ShortestPath** operator finds one or all shortest paths between two previously matches node variables.

Example 495. ShortestPath

Query

```
PROFILE
MATCH
  (andy:Person {name: 'Andy'}),
  (mattias:Person {name: 'Mattias'}),
  p = shortestPath((andy)-[*]-(mattias))
RETURN p
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator Details						Estimated Rows
Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline	
1	0			1/0	0.241	1
		+				
+ProduceResults	p					
1	1	1424			In Pipeline 1	1
		+				
+ShortestPath	p = (andy)-[anon_0*]-(mattias)					
1	1	120		1/1	0.308	1
+MultiNodeIndexSeek	RANGE INDEX andy:Person(name) WHERE name = \$autostring_0,					1
1	4	120		1/1	0.308	In Pipeline 0

Total database accesses: 5, total allocated memory: 1488

Empty Row

The `EmptyRow` operator returns a single row with no columns.

Example 496. EmptyRow

Query

```
PROFILE
CYpher runtime=slotted
FOREACH (value IN [1,2,3] | MERGE (:Person {age: value}))
```

Query Plan

Planner COST

Runtime SLOTTED

Runtime version 5.4

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.age = value	1	0	184	
2/0					
+NodeByLabelScan	anon_0:Person	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 497. ProcedureCall

Query

```
PROFILE
CALL db.labels() YIELD label
RETURN *
ORDER BY label
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	(Bytes)	Page Cache Hits/Misses	Time (ms)	Estimated Rows	Rows	DB Hits	Memory
					Ordered by		Pipeline	
+ProduceResults	label	0/0	0.091		10	4	0	
+Sort	label ASC	0/0	0.178	label ASC	10	4	0	
536					In Pipeline 1			
+ProcedureCall	db.labels() :: (label :: STRING?)				10	4		
					Fused in Pipeline 0			

Total database accesses: ?, total allocated memory: 600

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 498. CacheProperties

Query

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

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline	Estimated Rows	Rows	DB Hits
+ProduceResults	location, name					15	15	0
+Projection	cache[l.name] AS location, p.name AS name					15	15	30
+Filter	p:Person					15	15	30
+Expand(All)	(l)-[:WORKS_IN]-(p)					15	15	16
+CacheProperties	cache[l.name]					10	10	10
+NodeByLabelScan	l:Location	120	5/0	0,422	Fused in Pipeline 0	10	10	35

Total database accesses: 157, total allocated memory: 200

Create (nodes and relationships)

The `Create` operator is used to create nodes and relationships.

Example 499. Create

Query

```
PROFILE
CREATE
  (max:Person {name: 'Max'}),
  (chris:Person {name: 'Chris'})
CREATE (max)-[:FRIENDS_WITH]->(chris)
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details				
Estimated Rows	Rows	DB Hits	Page Cache Hits/Misses	Time (ms)	Pipeline
+ProduceResults	0	0			
+EmptyResult	0	0			
+Create	1	7	0/0 0.000	Fused in Pipeline 0 (max)-[anon_0:FRIENDS_WITH]->(chris)	

Total database accesses: 7, total allocated memory: 184

Delete (nodes and relationships)

The **Delete** operator is used to delete a node or a relationship.

Example 500. Delete

Query

```
PROFILE
MATCH (me:Person {name: 'me'})-[w:WORKS_IN {duration: 190}]->(london:Location {name: 'London'})
DELETE w
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator		Details			
Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)
+ProduceResults	0	0	0		
+EmptyResult	0	0	0		
+Delete	1	1	w		
+Eager	1	0	112	1/0	0.247 Fused in Pipeline 1
+Filter	1	5			london.name = \$autostring_2 AND w.duration = \$autoint_1 AND london:Location
+Expand(All)	1	5			(me)-[w:WORKS_IN]->(london)
+NodeIndexSeek	1	2	120	4/1	0.447 Fused in Pipeline 0

Total database accesses: 13, total allocated memory: 216

Detach Delete

The **DetachDelete** operator is used in all queries containing the **DETACH DELETE** clause, when deleting

nodes and their relationships.

Example 501. DetachDelete

Query

```
PROFILE  
MATCH (p:Person)  
DETACH DELETE p
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline
+ProduceResults		14	0	0				
+EmptyResult		14	0	0				
+DetachDelete	p	14	14	41				
+NodeByLabelScan	p:Person	14	14	35		120		
21/0	12,439	Fused in Pipeline 0						

Total database accesses: 112, total allocated memory: 200

Set Labels

The `SetLabels` operator is used when setting labels on a node.

Example 502. SetLabels

Query

```
PROFILE  
MATCH (n)  
SET n:Person
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline
+ProduceResults		35	0	0	0			
+EmptyResult		35	0	0	0			
+SetLabels	n:Person	35	35	22	0			
+AllNodesScan	n	35	35	36	120			
3/0	0.873	Fused in Pipeline 0						

Total database accesses: 58, total allocated memory: 184

Remove Labels

The `RemoveLabels` operator is used when deleting labels from a node.

Example 503. RemoveLabels

Query

```
PROFILE  
MATCH (n)  
REMOVE n:Person
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline
+ProduceResults		35	0	0	0			
+EmptyResult		35	0	0	0			
+RemoveLabels	n:Person	35	35	15	15			
+AllNodesScan	n	35	35	36	120			
3/0	0.765	Fused in Pipeline 0						

Total database accesses: 51, total allocated memory: 184

Set Node Properties From Map

The `SetNodePropertiesFromMap` operator is used when setting properties from a map on a node.

Example 504. SetNodePropertiesFromMap

Query

```
PROFILE
MATCH (n)
SET n = {weekday: 'Monday', meal: 'Lunch'}
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details					Estimated Rows
	Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	
+ProduceResults	0	0				35
+EmptyResult	0	0				35
+SetNodePropertiesFromMap	35	105				35
+AllNodesScan	35	36	120	5/0	3.954	Fused in Pipeline 0

Total database accesses: 141, total allocated memory: 184

Set Relationship Properties From Map

The `SetRelationshipPropertiesFromMap` operator is used when setting properties from a map on a relationship.

Example 505. SetRelationshipPropertiesFromMap

Query

```
PROFILE  
MATCH (n)-[r]->(m)  
SET r = {weight: 5, unit: 'kg'}
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator Details						Estimated Rows
Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline	
+ +ProduceResults	0 0 28					
		+-----+-----+				
+ +EmptyResult	0 0 28					
		+-----+-----+				
+ +SetRelationshipPropertiesFromMap	r = {weight: \$autoint_0, unit: \$autostring_1} 28					
28 84		+-----+-----+				
+ +DirectedAllRelationshipsScan	(n)-[r]->(m) 28					
28 28 120 5/0 15.278 Fused in Pipeline 0						

Total database accesses: 112, total allocated memory: 184

Set Property

The **SetProperty** operator is used when setting a property on a node or relationship.

Example 506. SetProperty

Query

```
PROFILE  
MATCH (n)  
SET n.checked = true
```

Query Plan

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator	Details	Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache
Hits/Misses	Time (ms)	Pipeline				
+ProduceResults		35	0	0		
+EmptyResult		35	0	0		
+SetProperty	n.checked = true	35	35	70		
+AllNodesScan	n	35	35	36	120	
3/0	0.753	Fused in Pipeline 0				

Total database accesses: 106, total allocated memory: 184

Create Constraint

The `CreateConstraint` operator creates a constraint.

This constraint can have any of the available constraint types:

- Property uniqueness constraints
- Property existence constraints Enterprise edition
- Node key constraints Enterprise edition

The following query will create a property uniqueness constraint with the name `uniqueness` on the `name` property of nodes with the `Country` label.

Example 507. CreateConstraint

Query

```
PROFILE
CREATE CONSTRAINT uniqueness
FOR (:Country) REQUIRE c.name IS UNIQUE
```

Query Plan

```
Planner ADMINISTRATION
Runtime SCHEMA
Runtime version 5.4
+-----+-----+
| Operator      | Details          |
+-----+-----+
| +CreateConstraint | CONSTRAINT uniqueness FOR (:Country) REQUIRE (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 property uniqueness constraint with the name `uniqueness` on the `name` property of nodes with the `Country` label only if no constraint named `uniqueness` or property uniqueness constraint on `(:Country {name})` already exists.

Example 508. DoNothingIfExists(CONSTRAINT)

Query

```
PROFILE
CREATE CONSTRAINT uniqueness IF NOT EXISTS
FOR (c:Country) REQUIRE c.name is UNIQUE
```

Query Plan

Planner ADMINISTRATION

Runtime SCHEMA

Runtime version 5.4

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: ?

Drop Constraint

The `DropConstraint` operator removes a constraint using the name of the constraint, no matter the type.

Example 509. DropConstraint

Query

```
PROFILE
DROP CONSTRAINT name
```

Query Plan

Planner ADMINISTRATION

Runtime SCHEMA

Runtime version 5.4

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 510. ShowConstraints

Query

```
PROFILE  
SHOW CONSTRAINTS
```

Query Plan

Planner COST

Runtime SLOTTED

Runtime version 5.4

Operator	Details	Estimated
Rows	Rows	DB Hits
+ProduceResults	id, name, type, entityType, labelsOrTypes, properties, ownedIndex	
10	3	0
		0/0
+ShowConstraints	allConstraints, defaultColumns	
10	3	2
		0/0

Total database accesses: 2, total allocated memory: 64

Create Index

The `CreateIndex` operator creates an index.

This index can either be a fulltext, point, range, text, or lookup index.

Example 511. CreateIndex

The following query will create an index with the name `my_index` on the `name` property of nodes with the `Country` label.

Query

```
PROFILE  
CREATE INDEX my_index  
FOR (c:Country) ON (c.name)
```

Query Plan

```
Planner ADMINISTRATION  
  
Runtime SCHEMA  
  
Runtime version 5.4  
  
+-----+  
| Operator | Details |  
+-----+  
| +CreateIndex | RANGE 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 512. DoNothingIfExists(INDEX)

Query

```
PROFILE
CREATE INDEX my_index IF NOT EXISTS
FOR ()-[k:KNOWS]-() ON (k.since)
```

Query Plan

Planner ADMINISTRATION

Runtime SCHEMA

Runtime version 5.4

Operator	Details
+CreateIndex	RANGE INDEX my_index FOR ()-[k:KNOWS]-() ON (since)
+DoNothingIfExists(INDEX)	RANGE INDEX my_index FOR ()-[k:KNOWS]-() ON (since)

Total database accesses: ?

Drop Index

The `DropIndex` operator removes an index using the name of the index.

Example 513. DropIndex

Query

```
PROFILE
DROP INDEX name
```

Query Plan

Planner ADMINISTRATION

Runtime SCHEMA

Runtime version 5.4

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 514. ShowIndexes

Query

```
PROFILE  
SHOW INDEXES
```

Query Plan

```
Planner COST  
  
Runtime SLOTTED  
  
Runtime version 5.4  
  
+-----  
+-----  
+-----+-----+-----+  
| Operator      | Details  
| Estimated Rows | Rows | DB Hits | Page Cache Hits/Misses |  
+-----  
+-----+-----+-----+  
| +ProduceResults | id, name, state, populationPercent, type, entityType, labelsOrTypes, properties,  
indexProvider, | 10 | 9 | 0 | 0/0 |  
| |           | owningConstraint |  
| |           |           |           |  
+-----  
+-----+-----+-----+  
| +ShowIndexes   | allIndexes, defaultColumns  
| 10 | 9 | 2 | 0/0 |  
+-----  
+-----+-----+-----+  
  
Total database accesses: 2, 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 515. ShowFunctions

Query

```
PROFILE  
SHOW FUNCTIONS
```

Query Plan

Planner COST

Runtime SLOTTED

Runtime version 5.4

Operator	Details	Estimated Rows	Rows	DB
Hits Page Cache Hits/Misses				
+ProduceResults name, category, description	0/0		10 147	
+ShowFunctions allFunctions, functionsForUser(all), defaultColumns	0/0		10 147	

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 516. ShowProcedures

Query

```
PROFILE  
SHOW PROCEDURES
```

Query Plan

```
Planner COST  
  
Runtime SLOTTED  
  
Runtime version 5.4  
  
+-----+-----+-----+-----+  
+-----+-----+-----+-----+  
| Operator      | Details          | Estimated Rows | Rows | DB Hits | Page  
Cache Hits/Misses |  
+-----+-----+-----+-----+  
+-----+-----+-----+-----+  
| +ProduceResults | name, description, mode, worksOnSystem |           10 |    55 |       0 |  
0/0 |  
| |  
+-----+-----+-----+-----+  
| +ShowProcedures | proceduresForUser(all), defaultColumns |           10 |    55 |       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 517. ShowTransactions

Query

```
PROFILE  
SHOW TRANSACTIONS
```

Query Plan

```
Planner COST  
  
Runtime SLOTTED  
  
Runtime version 5.4  
  
+-----  
+-----  
+-----+-----+-----+  
| Operator      | Details  
| Estimated Rows | Rows | DB Hits | Page Cache Hits/Misses |  
+-----  
+-----+-----+-----+  
| +ProduceResults | database, transactionId, currentQueryId, connectionId, clientAddress, username,  
currentQuery, |          10 |    1 |     0 |          0/0 |  
| |           | startTime, status, elapsedTime  
| |           |           |           |  
| |           |           |           |  
+-----+-----+-----+  
| +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 518. TerminateTransactions

Query

```
PROFILE  
TERMINATE TRANSACTIONS 'database-transaction-123'
```

Query Plan

Planner COST

Runtime SLOTTED

Runtime version 5.4

Operator	Details	Estimated Rows
Rows	DB Hits Page Cache Hits/Misses	
+ProduceResults	transactionId, username, message 0/0	10
+TerminateTransactions	defaultColumns, transactions(database-transaction-123) 0/0	10
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 519. Query evaluated with the fast algorithm

This query can be evaluated with the fast algorithm — there are no predicates that need to see the whole path before being evaluated.

Query

```
MATCH
  (KevinB:Person {name: 'Kevin Bacon'}),
  (Al:Person {name: 'Al Pacino'}),
  p = shortestPath((KevinB)-[:ACTED_IN*]-(Al))
  WHERE all(r IN relationships(p) WHERE r.role IS NOT NULL)
RETURN p
```

Query plan

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 128

+-----+
+-----+
+-----+-----+-----+-----+
| Operator          | Details
| Estimated Rows  | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline
| +-----+
| +ProduceResults | p   |
|     2 | 1 | 0 |           |           | 1/0 | 0.252 |
| |
| +-----+
| +ShortestPath    | p = (KevinB)-[anon_0:ACTED_IN*]-(Al) WHERE all(r IN relationships(p) WHERE
r.role IS NOT NULL) |
|     2 | 1 | 23 |           | 1688 |           |
| In Pipeline 1 |
| |
| +-----+
| +MultiNodeIndexSeek | RANGE INDEX KevinB:Person(name) WHERE name = $autostring_0,
|     2 | 1 | 4 |           | 120 |           | 1/1 | 0.916 | In Pipeline
0 |
|           | RANGE INDEX Al:Person(name) WHERE name = $autostring_1
|           |           |           |           |           |
| |
| +-----+
| +-----+
| +-----+-----+-----+-----+
Total database accesses: 27, total allocated memory: 1752
```

Shortest path — additional predicate checks on the paths

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

Example 520. 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

```
Planner COST
Runtime PIPELINED
Runtime version 5.4
Batch size 1024

+-----+
+-----+-----+-----+
+-----+-----+-----+-----+
| Operator          | Details                                | Estimated
| Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
| +ProduceResults   | p
1 | 1 | 0 |           |           |           |           |
| | | +-----+-----+
| | +AntiConditionalApply
1 | 1 | 0 |           | 41464 |           | 0/0 | 0.332 | Fused in Pipeline 6 |
| | \ +-----+-----+
+-----+-----+-----+-----+
+-----+-----+-----+-----+
| | +Top
2 | 0 | 0 |           | 4280 |           | 0/0 | 0.000 | In Pipeline 5
| | | +-----+-----+
+-----+-----+-----+-----+
+-----+-----+-----+-----+
| | +Projection
7966 | 0 | 0 |           | length(p) AS anon_1
| | | +-----+-----+
+-----+-----+-----+-----+
| | +Filter
7966 | 0 | 0 |           | length(p) > $autoint_2
| | | +-----+-----+
+-----+-----+-----+-----+
| | +Projection
26554 | 0 | 0 |           | (KevinB)-[anon_0*]-(Al) AS p
| | | +-----+-----+
+-----+-----+-----+-----+
| | +VarLengthExpand(Into)
26554 | 0 | 0 |           | (KevinB)-[anon_0*]-(Al)
| | | +-----+-----+
+-----+-----+-----+-----+
| | +Argument
2 | 0 | 0 |           | KevinB, Al
| | | 0 |           |           | 0/0 | 0.000 | Fused in Pipeline 4 |
| | | +-----+-----+
```

```

+---+
| +Apply
2 |   1 |     0 |           |                               0/0 |    0.026 |
| |\ \
+---+-----+-----+-----+
+---+
| | +Optional          | KevinB, Al
2 |   1 |     0 |           4840 |               0/0 |    0.134 | In Pipeline 3
| | |
+---+-----+-----+-----+
+---+
| | +ShortestPath      | p = (KevinB)-[anon_0*]-(Al) WHERE length(p) > $autoInt_2
1 |   1 |     1 |           1760 |           |           | In Pipeline 2
| | |
+---+-----+-----+-----+
+---+
| | +Argument          | KevinB, Al
2 |   1 |     0 |           24680 |               0/0 |    0.056 | In Pipeline 1
| |
+---+-----+-----+-----+
+---+
| +MultiNodeIndexSeek | RANGE INDEX KevinB:Person(name) WHERE name = $autoString_0,
2 |   1 |     4 |           120 |           2/0 |    0.644 | In Pipeline 0
|           | RANGE INDEX Al:Person(name) WHERE name = $autoString_1
|           |
+---+-----+-----+-----+
+---+

```

Total database accesses: 5, total allocated memory: 50152

The way the bigger exhaustive query plan works is by using `Apply/Optional` to ensure that when the fast algorithm does not find any results, a `null` result is generated instead of simply stopping the result stream. On top of this, the planner will issue an `AntiConditionalApply`, which will run the exhaustive search if the path variable is pointing to `null` instead of a path.

An `ErrorPlan` operator will appear in the execution plan in cases where:

- `dbms.cypher.forbid_exhaustive_shortestpath` is set to `true`.
- The fast algorithm is not able to find the shortest path.

Example 521. 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

Planner COST

Runtime PIPELINED

Runtime version 5.4

Batch size 128

Operator Details						Estimated Rows
Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline	
+ProduceResults p	1	0		1/0 0.353		1
+Filter length(p) > \$autoint_2	1	0		0/0 0.255		1
+ShortestPath p = (KevinB)-[anon_0*]-(Al)	1	1	1760		In Pipeline 1	2
+MultiNodeIndexSeek RANGE INDEX KevinB:Person(name) WHERE name = \$autostring_0,	1	4	120	2/0 0.371	In Pipeline 0	2
				RANGE INDEX Al:Person(name) WHERE name = \$autostring_1		

Total database accesses: 5, total allocated memory: 1824

Deprecations, additions, and compatibility

Cypher is a language that is constantly evolving. New features are added to the language continuously, and occasionally, some features become deprecated and are subsequently removed.

This section lists all of the features that have been removed, deprecated, added, or extended in different Cypher versions. Replacement syntax for deprecated and removed features are also indicated.

Version 5.3

Updated features

Feature	Details
Functionality Updated SHOW DATABASES	Changes to the visibility of databases hosted on offline servers. For such databases: <ul style="list-style-type: none">The <code>address</code> column will return <code>NULL</code>.The <code>currentStatus</code> column will return <code>unknown</code>.The <code>statusMessage</code> will return <code>Server is unavailable</code>.
Functionality Updated EXISTS { ... }	An <code>EXISTS</code> subquery now supports any non-writing query. For example, it now supports <code>UNION</code> and <code>CALL</code> clauses.
Functionality Updated COUNT { ... }	A <code>COUNT</code> subquery now supports any non-writing query. For example, it now supports <code>UNION</code> and <code>CALL</code> clauses.
Syntax Updated SHOW <code>UNIQUE[NESS]</code> CONSTRAINTS	The property uniqueness constraint type filter now allow both <code>UNIQUE</code> and <code>UNIQUENESS</code> keywords.

Version 5.2

Deprecated features

Feature	Details
Syntax Deprecated <code>MATCH ()-[r*]-()-[r*]-()</code> <code>MATCH p = ()-[r*]-(), q = ()-[r*]-()</code> <code>MATCH ()-[r*]-() MATCH ()-[r*]-()</code>	The use of the same relationship variable for multiple variable length relationships is deprecated.

Updated features

Feature	Details
Syntax Updated <code>CREATE COMPOSITE DATABASE name OPTIONS {}</code>	Creating composite databases now allows for an empty options clause. There are no applicable option values for composite databases.
Functionality New <code>DRYRUN REALLOCATE DEALLOCATE DATABASES FROM <serverId></code>	To preview of the result of either <code>REALLOCATE</code> or <code>DEALLOCATE</code> without executing, prepend the command with <code>DRYRUN</code> .

Version 5.1

Deprecated features

Feature	Details
Functionality Deprecated <code>CREATE TEXT INDEX ... OPTIONS {indexProvider: 'text-1.0'}</code>	The text index provider <code>text-1.0</code> is deprecated and replaced by <code>text-2.0</code> .

Updated features

Feature	Details
Functionality Updated <pre>CREATE TEXT INDEX ... OPTIONS {indexProvider: 'text-2.0'}</pre>	A new text index provider is available, <code>text-2.0</code> . This is also the default provider if none is given.

Version 5.0

Removed features

Feature	Details
Syntax Removed <pre>SHOW EXISTS CONSTRAINTS</pre>	Replaced by: <pre>SHOW [PROPERTY] EXIST[ENCE] CONSTRAINTS</pre>
<pre>SHOW NODE EXISTS CONSTRAINTS</pre>	<pre>SHOW NODE [PROPERTY] EXIST[ENCE] CONSTRAINTS</pre>
<pre>SHOW RELATIONSHIP EXISTS CONSTRAINTS</pre>	<pre>SHOW RELATIONSHIP [PROPERTY] EXIST[ENCE] CONSTRAINTS</pre>
Syntax Removed <pre>SHOW INDEXES BRIEF</pre>	Replaced by: <pre>SHOW INDEXES</pre>
<pre>SHOW CONSTRAINTS BRIEF</pre>	<pre>SHOW CONSTRAINTS</pre>
Syntax Removed <pre>SHOW INDEXES VERBOSE</pre>	Replaced by: <pre>SHOW INDEXES YIELD *</pre>
<pre>SHOW CONSTRAINTS VERBOSE</pre>	<pre>SHOW CONSTRAINTS YIELD *</pre>
Functionality Removed <pre>DROP INDEX ON :Label(prop)</pre>	Replaced by: <pre>DROP INDEX name</pre>

Feature	Details
<p>Functionality Removed</p> <pre>DROP CONSTRAINT ON (n:Label) ASSERT (n.prop) IS NODE KEY</pre>	<p>Replaced by:</p> <pre>DROP CONSTRAINT name</pre>
<pre>DROP CONSTRAINT ON (n:Label) ASSERT (n.prop) IS UNIQUE</pre>	
<pre>DROP CONSTRAINT ON (n:Label) ASSERT exists(n.prop)</pre>	
<pre>DROP CONSTRAINT ON ()-[r:Type]-() ASSERT exists (r.prop)</pre>	
<p>Syntax Removed</p> <pre>CREATE INDEX ON :Label(prop)</pre>	<p>Replaced by:</p> <pre>CREATE INDEX FOR (n:Label) ON (n.prop)</pre>
<p>Syntax Removed</p> <pre>CREATE CONSTRAINT ON ... ASSERT ...</pre>	<p>Replaced by:</p> <pre>CREATE CONSTRAINT FOR ... REQUIRE ...</pre>
<p>Functionality Removed</p> <pre>CREATE BTREE INDEX ...</pre>	<p>B-tree indexes are removed.</p> <p>B-tree indexes used for string queries are replaced by:</p> <pre>CREATE TEXT INDEX ...</pre>
<p>Functionality Removed</p> <pre>CREATE INDEX ... OPTIONS "{" btree-option: btree-value[, ...] "}"</pre>	<p>B-tree indexes used for spatial queries are replaced by:</p> <pre>CREATE POINT INDEX ...</pre> <p>B-tree indexes used for general queries or property value types are replaced by:</p> <pre>CREATE [RANGE] INDEX ...</pre>
	<p>These new indexes may be combined for multiple use cases.</p>

Feature	Details
Functionality Removed <pre>SHOW BTREE INDEXES</pre>	B-tree indexes are removed. Replaced by: <pre>SHOW {POINT RANGE TEXT} INDEXES</pre>
Functionality Removed <pre>USING BTREE INDEXES</pre>	B-tree indexes are removed. Replaced by: <pre>USING {POINT RANGE TEXT} INDEX</pre>
Functionality Removed <pre>CREATE CONSTRAINT ... OPTIONS "{" btree-option: btree-value[, ...] "}"</pre>	Node key and property uniqueness constraints backed by B-tree indexes are removed. Replaced by: <pre>CREATE CONSTRAINT ...</pre>
	Constraints used for string properties require an additional text index to cover the string queries properly. Constraints used for point properties require an additional point index to cover the spatial queries properly.
Functionality Removed <pre>SHOW INDEXES YIELD uniqueness</pre>	The uniqueness output has been removed along with the concept of index uniqueness, as it actually belongs to the constraint and not the index. The new column owningConstraint was introduced to indicate whether an index belongs to a constraint or not.
Functionality Removed <pre>SHOW CONSTRAINTS YIELD ownedIndexId</pre>	The ownedIndexId output has been removed and replaced by the new ownedIndex column.
Syntax Removed For privilege commands: <pre>ON DEFAULT DATABASE</pre>	Replaced by: <pre>ON HOME DATABASE</pre>

Feature	Details
Syntax Removed For privilege commands: <code>ON DEFAULT GRAPH</code>	Replaced by: <code>ON HOME GRAPH</code>
Functionality Removed <code>SHOW TRANSACTIONS YIELD allocatedBytes</code>	The <code>allocatedBytes</code> output has been removed, because it was never tracked and thus was always 0.
Syntax Removed <code>exists(prop)</code>	Replaced by: <code>prop IS NOT NULL</code>
Syntax Removed <code>NOT exists(prop)</code>	Replaced by: <code>prop IS NULL</code>
Syntax Removed <code>0...</code>	Replaced by <code>0o....</code>
Syntax Removed <code>0X...</code>	Only <code>0x....</code> (lowercase x) is supported.
Syntax Removed <code>MATCH ()-[r]-() RETURN [()-[r]-()-[r]-() r] AS rs</code>	Remaining support for repeated relationship variables is removed.
Syntax Removed <code>WHERE [1, 2, 3]</code>	Automatic coercion of a list to a boolean is removed. Replaced by: <code>WHERE NOT isEmpty([1, 2, 3])</code>

Feature	Details
Functionality Removed	Replaced by: <code>point.distance(n.prop, point({x:0, y:0}))</code>
Functionality Removed	The ability to use operators <code><</code> , <code>≤</code> , <code>></code> , and <code>≥</code> on spatial points is removed. Instead, use: <code>point.withinBBox(point({x:1, y:1}), point({x:0, y:0}), point({x:2, y:2}))</code>
Syntax Removed	Replaced by: <code>CALL { ... } IN TRANSACTIONS</code>
Syntax Removed	It is no longer allowed to have <code>CREATE</code> clauses in which a variable introduced in the pattern is also referenced from the same pattern.
Syntax Removed	Unaliased expressions are no longer supported in subquery <code>RETURN</code> clauses. Replaced by: <code>CALL { RETURN 1 AS one }</code>
Syntax Removed	Pattern expressions producing lists of paths are no longer supported, but they can still be used as existence predicates, for example in <code>WHERE</code> clauses. Instead, use a pattern comprehension: <code>MATCH (a) RETURN [p=(a)--() p]</code>

Feature	Details
<p>Functionality Removed</p> <pre>MATCH (n) RETURN n.propertyName_1, n.propertyName_2 + count(*)</pre>	<p>Implied grouping keys are no longer supported. Only expressions that do not contain aggregations are still considered grouping keys. In expressions that contain aggregations, the leaves must be either:</p> <ul style="list-style-type: none"> • An aggregation • A literal • A parameter • A variable, ONLY IF it is either: 1) A projection expression on its own (e.g. the <code>n</code> in <code>RETURN n AS myNode, n.value + count(*)</code>) 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>) • 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>) • 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>)

Deprecated features

Feature	Details
<p>Syntax Deprecated</p> <pre>MATCH (n)-[r:REL]->(m) SET n=r</pre>	<p>Use the <code>properties()</code> function instead to get the map of properties of nodes/relationships that can then be used in a <code>SET</code> clause:</p> <pre>MATCH (n)-[r:REL]->(m) SET n=properties(r)</pre>
<p>Syntax Deprecated</p> <pre>MATCH (a), (b), allShortestPaths((a)-[r]->(b)) RETURN b</pre> <pre>MATCH (a), (b), shortestPath((a)-[r]->(b)) RETURN b</pre>	<p><code>shortestPath</code> and <code>allShortestPaths</code> without variable-length relationship are deprecated. Instead, use a <code>MATCH</code> with a <code>LIMIT</code> of 1 or:</p> <pre>MATCH (a), (b), shortestPath((a)-[r*1..1]->(b)) RETURN b</pre>

Feature	Details
<p>Syntax Deprecated</p> <pre>CREATE DATABASE databaseName.withDot ...</pre>	<p>Creating a database with unescaped dots in the name has been deprecated, instead escape the database name:</p> <pre>CREATE DATABASE `databaseName.withDot` ...</pre>
<p>Functionality Deprecated</p> <pre>(-)[:-A :B]->()</pre>	<p>Replaced by:</p> <pre>(-)[:-A B]->()</pre>

Updated features

Feature	Details
<p>Functionality Updated</p> <pre>CREATE INDEX ...</pre>	<p>The default index type is changed from B-tree to range index.</p>
<p>Functionality Updated</p> <pre>SHOW INDEXES</pre>	<p>The new column <code>owningConstraint</code> was added and will be returned by default from now on. It will list the name of the constraint that the index is associated with or <code>null</code>, in case it is not associated with any constraint.</p>
<p>Functionality Updated</p> <pre>SHOW CONSTRAINTS</pre>	<p>The new column <code>ownedIndex</code> was added and will be returned by default from now on. It will list the name of the index associated with the constraint or <code>null</code>, in case no index is associated with it.</p>

Feature	Details
Functionality Updated SHOW TRANSACTIONS YIELD *	<p>New columns for the current query are added:</p> <ul style="list-style-type: none"> • <code>currentQueryStartTime</code> • <code>currentQueryStatus</code> • <code>currentQueryActiveLockCount</code> • <code>currentQueryElapsedTime</code> • <code>currentQueryCpuTime</code> • <code>currentQueryWaitTime</code> • <code>currentQueryIdleTime</code> • <code>currentQueryAllocatedBytes</code> • <code>currentQueryPageHits</code> • <code>currentQueryPageFaults</code> <p>These columns are only returned in the full set (with YIELD) and not by default.</p>
Functionality Updated TERMINATE TRANSACTIONS <code>transaction-id[,...]</code> <code>[YIELD { * field[, ...] }]</code> <code>[ORDER BY field[, ...]]</code> <code>[SKIP n]</code> <code>[LIMIT n]</code> <code>[WHERE expression]</code> <code>[RETURN field[, ...] [ORDER BY field[, ...]] [SKIP n] [LIMIT n]]</code>	Terminate transaction now allows YIELD . The WHERE clause is not allowed on its own, as it is for SHOW , but needs the YIELD clause.
Functionality Updated SHOW TRANSACTIONS <code>[transaction-id[,...]]</code> TERMINATE TRANSACTIONS <code>transaction-id[,...]</code>	<code>transaction-id</code> now allows general expressions resolving to a string or a list of strings instead of just parameters.

Feature	Details
Functionality Updated <pre>SHOW TRANSACTIONS [transaction-id[, ...]] YIELD field[, ...] [ORDER BY field[, ...]] [SKIP n] [LIMIT n] [WHERE expression] TERMINATE TRANSACTIONS transaction-id[, ...] YIELD field[, ...] [ORDER BY field[, ...]] [SKIP n] [LIMIT n] [WHERE expression] RETURN field[, ...] [ORDER BY field[, ...]] [SKIP n] [LIMIT n]</pre>	<p>The <code>SHOW</code> and <code>TERMINATE TRANSACTIONS</code> commands can be combined in the same query. The query does not require a specific order and there can be zero or more of each command type, however at least one command is needed.</p> <p>When the command is not in standalone mode, the <code>YIELD</code> and <code>RETURN</code> clauses are mandatory. <code>YIELD *</code> is not allowed.</p> <p><code>transaction-id</code> is a comma-separated list of one or more quoted strings. It could also be an expression resolving to a string or a list of strings (for example the output column from <code>SHOW</code>).</p>
Functionality Updated <pre>GRANT EXECUTE BOOSTED PROCEDURE ... GRANT EXECUTE BOOSTED FUNCTION ...</pre>	<p>Not a syntax change but a semantic one. The <code>EXECUTE BOOSTED</code> privilege will no longer include an implicit <code>EXECUTE</code> privilege when granted. That means that to execute a procedure or a function with boosted privileges both <code>EXECUTE</code> and <code>EXECUTE BOOSTED</code> are needed.</p>
Functionality Updated <pre>[GRANT DENY] [IMMUTABLE] ...</pre>	<p>Privileges can be specified as <code>IMMUTABLE</code>, which means that they cannot be altered by users with Privilege Management. They can only be administered with auth disabled.</p>
Functionality Updated <pre>REVOKE [IMMUTABLE] ...</pre>	<p><code>IMMUTABLE</code> can now be specified with the <code>REVOKE</code> command to specify that only immutable privileges should be revoked.</p>

Feature	Details
<p>Functionality Updated</p> <pre>SHOW DATABASES</pre>	<p>Changes to the default columns in the result:</p> <ul style="list-style-type: none"> The <code>writer</code>, <code>type</code>, and <code>constituents</code> columns have been added. The values returned in the <code>role</code> column have changes to be just <code>primary</code>, <code>secondary</code>, or <code>unknown</code>. The <code>error</code> column has been renamed to <code>statusMessage</code>. <p>The following columns have been added to the full result set (with <code>YIELD</code>) and not by default:</p> <ul style="list-style-type: none"> <code>creationTime</code> <code>lastStartTime</code> <code>lastStopTime</code> <code>store</code> <code>currentPrimariesCount</code> <code>currentSecondariesCount</code> <code>requestedPrimariesCount</code> <code>requestedSecondariesCount</code>
<p>Functionality Updated</p> <pre>MATCH (n) RETURN CASE n.prop WHEN null THEN 'one' ELSE 'two' END</pre>	<p>Previously, if <code>n.prop</code> is <code>null</code>, '<code>one</code>' would be returned. Now, '<code>two</code>' is returned.</p> <p>This is a semantic change only. Since <code>null = null</code> returns <code>false</code> in Cypher, a <code>WHEN</code> expression no longer matches on <code>null</code>.</p> <p>If matching on <code>null</code> is required, please use <code>IS NULL</code> instead:</p> <pre>MATCH (n) RETURN CASE WHEN n.prop IS NULL THEN 'one' ELSE 'two' END</pre>

Feature	Details															
Functionality Updated <pre>RETURN round(val, precision)</pre>	<p>Rounding infinity and NaN values will now return the original value instead of returning an integer approximation for precision 0 and throwing an exception for precision > 0:</p> <table border="1"> <thead> <tr> <th></th> <th>old value</th> <th>new value</th> </tr> </thead> <tbody> <tr> <td>round(Inf)</td> <td>92233720368547 76000.0</td> <td>Inf</td> </tr> <tr> <td>round(Inf, 1)</td> <td>exception</td> <td>Inf</td> </tr> <tr> <td>round(NaN)</td> <td>0</td> <td>Nan</td> </tr> <tr> <td>round(Inf, 1)</td> <td>exception</td> <td>Nan</td> </tr> </tbody> </table> <p>To get an integer value use the <code>toInteger</code> function.</p>		old value	new value	round(Inf)	92233720368547 76000.0	Inf	round(Inf, 1)	exception	Inf	round(NaN)	0	Nan	round(Inf, 1)	exception	Nan
	old value	new value														
round(Inf)	92233720368547 76000.0	Inf														
round(Inf, 1)	exception	Inf														
round(NaN)	0	Nan														
round(Inf, 1)	exception	Nan														
Functionality Updated <pre>CREATE [OR REPLACE] ALIAS compositeDatabase.aliasName ... ALTER ALIAS compositeDatabase.aliasName DROP ALIAS compositeDatabase.aliasName</pre>	The alias commands can now handle aliases in composite databases.															
Syntax Updated <pre>SHOW ALIAS[ES] aliasName FOR DATABASE[S] SHOW ALIAS[ES] compositeDatabase.aliasName FOR DATABASE[S]</pre>	SHOW ALIAS now allows for easy filtering on alias name.															
Functionality Updated <pre>CREATE [OR REPLACE] ALIAS compositeDatabase.aliasName ... ALTER ALIAS compositeDatabase.aliasName DROP ALIAS compositeDatabase.aliasName</pre>	The alias commands can now handle aliases in composite databases.															
Syntax Updated <pre>SHOW ALIAS[ES] aliasName FOR DATABASE[S] SHOW ALIAS[ES] compositeDatabase.aliasName FOR DATABASE[S]</pre>	SHOW ALIAS now allows for easy filtering on alias name.															

New features

Feature	Details
<p>Functionality New</p> <pre><code>CREATE [OR REPLACE] COMPOSITE DATABASE databaseName [IF NOT EXISTS] [WAIT [n SEC[OND[S]]] NOWAIT] DROP COMPOSITE DATABASE databaseName [IF EXISTS] [DUMP DATA DESTROY DATA] [WAIT [n SEC[OND[S]]] NOWAIT]</code></pre>	New Cypher command for creating and dropping composite databases.
<p>Functionality New</p> <p>New privilege:</p> <pre><code>CREATE COMPOSITE DATABASE DROP COMPOSITE DATABASE COMPOSITE DATABASE MANAGEMENT</code></pre>	New privileges that allow a user to CREATE and/or DROP composite databases.
<p>Syntax Added</p> <pre><code>1_000_000, 0x_FF_FF, 0o_88_88</code></pre>	Cypher now supports number literals with underscores between digits.
<p>Functionality Added</p> <pre><code>isNaN(n.prop)</code></pre>	New function which returns whether the given number is NaN . NaN is a special floating point number defined in the Floating-Point Standard IEEE 754. This function was introduced since comparisons including NaN = NaN returns false.
<p>Functionality Added</p> <pre><code>Nan, Inf, Infinity</code></pre>	Cypher now supports float literals for the values Infinity and NaN . NaN defines a quiet not-a-number value and does not throw any exceptions in arithmetic operations. Both values are implemented according to the Floating-Point Standard IEEE 754.
<p>Functionality Added</p> <pre><code>COUNT { (n) WHERE n.foo = "bar" }</code></pre>	New expression which returns the number of results of a subquery.
<p>Functionality Added</p> <pre><code>CREATE DATABASE ... TOPOLOGY n PRIMAR{Y IES} [m SECONDAR{Y IES}]</code></pre>	New sub-clause for CREATE DATABASE , to specify the number of servers hosting a database, when creating a database in cluster environments.

Feature	Details
Functionality Added	New sub-clause for ALTER DATABASE , which allows modifying the number of servers hosting a database in cluster environments. <code>ALTER DATABASE ... SET TOPOLOGY n PRIMAR{Y IES} [m SECONDAR{Y IES}]</code>
Functionality Added	New Cypher command for enabling servers. <code>ENABLE SERVER ...</code>
Functionality Added	New Cypher command for setting options for a server. <code>ALTER SERVER ... SET OPTIONS ...</code>
Functionality Added	New Cypher command for changing the name of a server. <code>RENAME SERVER ... TO ...</code>
Functionality Added	New Cypher command for re-balancing what servers host which databases. <code>REALLOCATE DATABASES</code>
Functionality Added	New Cypher command for moving all databases from servers. <code>DEALLOCATE DATABASE[S] FROM SERVER[S] ...</code>
Functionality Added	New Cypher command for dropping servers. <code>DROP SERVER ...</code>
Functionality Added	New Cypher command for listing servers. <code>SHOW SERVERS</code>

Feature	Details
Functionality New New privileges: <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;">SERVER MANAGEMENT</div> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;">SHOW SERVERS</div>	New privileges that allow a user to create, modify, reallocate, deallocate, drop and list servers.
Syntax New <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;">MATCH (n: A&(B C)&!D)</div>	New concise syntax for expressing predicates for which labels a node may have, referred to as label expression.
Syntax New <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;">MATCH ()-[r:(!A&!B)]->()</div>	New concise syntax for expressing predicates for which relationship types a relationship may have, referred to as relationship type expression.
Syntax New <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;">MATCH ()-[r:R {prop1: 42} WHERE r.prop2 > 42]->()</div>	New syntax that enables inlining of WHERE clauses inside relationship patterns.

Version 4.4

Deprecated features

Feature	Details
Functionality Deprecated	<p>Implied grouping keys are deprecated. Only expressions that do not contain aggregations are still considered grouping keys. In expressions that contain aggregations, the leaves must be either:</p> <ul style="list-style-type: none"> • An aggregation • A literal • A parameter • A variable, ONLY IF it is either: <ol style="list-style-type: none"> 1) A projection expression on its own (e.g. the <code>n</code> in <code>RETURN n AS myNode, n.value + count(*)</code>) 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>) • 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>) • 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>)
Syntax Deprecated	<p>Replaced by:</p> <pre>CALL { ... } IN TRANSACTIONS</pre>
Syntax Deprecated	<p><code>CREATE</code> clauses in which a variable introduced in the pattern is also referenced from the same pattern are deprecated.</p>
Syntax Deprecated	<p>Replaced by:</p> <pre>CREATE CONSTRAINT FOR ... REQUIRE ...</pre>

Feature	Details
<p>Functionality Deprecated</p> <pre>CREATE BTREE INDEX ...</pre>	<p>B-tree indexes are deprecated.</p> <p>B-tree indexes used for string queries are replaced by:</p> <pre>CREATE TEXT INDEX ...</pre>
<p>Functionality Deprecated</p> <pre>CREATE INDEX ... OPTIONS "{" btree-option: btree-value[, ...] "}"</pre>	<p>B-tree indexes used for spatial queries are replaced by:</p> <pre>CREATE POINT INDEX ...</pre> <p>B-tree indexes used for general queries or property value types are replaced by:</p> <pre>CREATE RANGE INDEX ...</pre> <p>These new indexes may be combined for multiple use cases.</p>
<p>Functionality Deprecated</p> <pre>SHOW BTREE INDEXES</pre>	<p>B-tree indexes are deprecated.</p> <p>Replaced by:</p> <pre>SHOW {POINT RANGE TEXT} INDEXES</pre>
<p>Functionality Deprecated</p> <pre>USING BTREE INDEX</pre>	<p>B-tree indexes are deprecated.</p> <p>Replaced by:</p> <pre>USING {POINT RANGE TEXT} INDEX</pre>

Feature	Details
<p>Functionality Deprecated</p> <pre>CREATE CONSTRAINT ... OPTIONS "{" btree-option: btree-value[, ...] "}"</pre>	<p>Node key and property uniqueness constraints with B-tree options are deprecated.</p> <p>Replaced by:</p> <pre>CREATE CONSTRAINT ... OPTIONS "{" range-option: range-value[, ...] "}"</pre> <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.</p>
<p>Functionality Deprecated</p> <pre>distance(n.prop, point({x:0, y:0}))</pre>	<p>Replaced by:</p> <pre>point.distance(n.prop, point({x:0, y:0}))</pre>
<p>Functionality Deprecated</p> <pre>point({x:0, y:0}) <= point({x:1, y:1}) <= point({x:2, y:2})</pre>	<p>The ability to use the inequality operators <code><</code>, <code><=</code>, <code>></code>, and <code>>=</code> on spatial points is deprecated. Instead, use:</p> <pre>point.withinBBox(point({x:1, y:1}), point({x:0, y:0}), point({x:2, y:2}))</pre>
<p>Functionality Deprecated</p> <pre>MATCH (n) RETURN CASE n.prop WHEN null THEN 'one' ELSE 'two' END</pre>	<p>Currently, if <code>n.prop</code> is <code>null</code>, 'one' would be returned. Since <code>null = null</code> returns <code>false</code> in Cypher, a <code>WHEN</code> expression will no longer match in future versions.</p> <p>Please use <code>IS NULL</code> instead:</p> <pre>MATCH (n) RETURN CASE WHEN n.prop IS NULL THEN 'one' ELSE 'two' END</pre>

New features

Feature	Details
<p>Functionality New</p> <pre>CALL { ... } IN TRANSACTIONS</pre>	New clause for evaluating a subquery in separate transactions. Typically used when modifying or importing large amounts of data. See CALL { ... } IN TRANSACTIONS .
<p>Syntax New</p> <pre>CREATE CONSTRAINT FOR ... REQUIRE ...</pre>	New syntax for creating constraints, applicable to all constraint types.
<p>Functionality New</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>	Property uniqueness constraints now allow multiple properties, ensuring that the combination of property values are unique.
<p>Functionality New Deprecated</p> <pre>DROP CONSTRAINT ON (n:LabelName) ASSERT (n.propertyName_1, ..., n.propertyName_n) IS UNIQUE</pre>	<p>Property uniqueness constraints now allow multiple properties.</p> <p>Replaced by:</p> <pre>DROP CONSTRAINT name [IF EXISTS]</pre>
<p>Syntax New</p> <pre>CREATE CONSTRAINT [constraint_name] [IF NOT EXISTS] FOR ... REQUIRE ... IS NOT NULL OPTIONS "{" "}"</pre>	Existence constraints now allow an OPTIONS map, however, at this point there are no available values for the map.
<p>Functionality New</p> <pre>CREATE LOOKUP INDEX [index_name] [IF NOT EXISTS] FOR ... ON ... OPTIONS "{" option: value[, ...] "}"</pre>	Token lookup indexes now allow an OPTIONS map to specify the index provider.
<p>Functionality New</p> <pre>CREATE TEXT INDEX ...</pre>	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.

Feature	Details
<p>Functionality New</p> <pre>CREATE RANGE INDEX ...</pre>	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>Functionality New</p> <pre>CREATE CONSTRAINT ... OPTIONS "{" indexProvider: 'range-1.0' "}"</pre>	Allows creating node key and property uniqueness constraints backed by range indexes by providing the range index provider in the OPTIONS map.
<p>Functionality New</p> <pre>CREATE POINT INDEX ...</pre>	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>Syntax New</p> <p>New privilege:</p> <pre>IMPERSONATE</pre>	New privilege that allows a user to assume privileges of another one.
<p>Functionality New</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 transaction-id is a comma-separated list of one or more quoted strings, a string parameter, or a list parameter.</p> <p>This replaces the procedures <code>dbms.listTransactions</code> and <code>dbms.listQueries</code>.</p>
<p>Functionality New</p> <pre>TERMINATE TRANSACTION[S] transaction-id[,...]</pre>	<p>Terminate transactions on the current server.</p> <p>The transaction-id is a comma-separated list of one or more quoted strings, a string parameter, or a list parameter.</p> <p>This replaces the procedures <code>dbms.killTransaction</code>, <code>dbms.killTransactions</code>, <code>dbms.killQuery</code>, and <code>dbms.killQueries</code>.</p>

Feature	Details
Functionality New <pre data-bbox="155 265 584 316">ALTER DATABASE ... [IF EXISTS] SET ACCESS {READ ONLY READ WRITE}</pre>	New Cypher command for modifying a database by changing its access mode.
Functionality New New privilege: <pre data-bbox="155 534 330 563">ALTER DATABASE</pre>	New privilege that allows a user to modify databases.
Functionality New New privilege: <pre data-bbox="155 774 389 804">SET DATABASE ACCESS</pre>	New privilege that allows a user to modify database access mode.
Functionality New <pre data-bbox="155 977 547 1028">CREATE ALIAS ... [IF NOT EXISTS] FOR DATABASE ...</pre>	New Cypher command for creating an alias for a database name. Remote aliases are only supported from version 4.4.8.
Functionality New <pre data-bbox="155 1201 484 1253">CREATE OR REPLACE ALIAS ... FOR DATABASE ...</pre>	New Cypher command for creating or replacing an alias for a database name. Remote aliases are only supported from version 4.4.8.
Functionality New <pre data-bbox="155 1432 484 1484">ALTER ALIAS ... [IF EXISTS] SET DATABASE ...</pre>	New Cypher command for altering an alias. Remote aliases are only supported from version 4.4.8.
Functionality New <pre data-bbox="155 1650 632 1680">DROP ALIAS ... [IF EXISTS] FOR DATABASE</pre>	New Cypher command for dropping a database alias.
Functionality New <pre data-bbox="155 1859 462 1888">SHOW ALIASES FOR DATABASE</pre>	New Cypher command for listing database aliases. Only supported since version 4.4.8.

Feature	Details
Functionality New New privilege: <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;">ALIAS MANAGEMENT</div>	New privilege that allows a user to create, modify, delete and list aliases. Only supported since version 4.4.8.
Functionality New New privilege: <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;">CREATE ALIAS</div>	New privilege that allows a user to create aliases. Only supported since version 4.4.8.
Functionality New New privilege: <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;">ALTER ALIAS</div>	New privilege that allows a user to modify aliases. Only supported since version 4.4.8.
Functionality New New privilege: <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;">DROP ALIAS</div>	New privilege that allows a user to delete aliases. Only supported since version 4.4.8.
Functionality New New privilege: <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;">SHOW ALIAS</div>	New privilege that allows a user to show aliases. Only supported since version 4.4.8.
Syntax New <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;">MATCH (n:N {prop1: 42}) WHERE n.prop2 > 42</div>	New syntax that enables inlining of WHERE clauses inside node patterns.

Version 4.3

Deprecated features

Feature	Details
<p>Syntax Deprecated</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>Syntax Deprecated</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>Syntax Deprecated</p> <pre>exists(prop)</pre>	<p>Replaced by:</p> <pre>prop IS NOT NULL</pre>
<p>Syntax Deprecated</p> <pre>NOT exists(prop)</pre>	<p>Replaced by:</p> <pre>prop IS NULL</pre>
<p>Syntax Deprecated</p> <p>BRIEF [OUTPUT] for SHOW INDEXES and SHOW CONSTRAINTS.</p>	<p>Replaced by default output columns.</p>
<p>Syntax Deprecated</p> <p>VERBOSE [OUTPUT] for SHOW INDEXES and SHOW CONSTRAINTS.</p>	<p>Replaced by:</p> <pre>YIELD *</pre>
<p>Syntax Deprecated</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>Syntax Deprecated</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>

Feature	Details
Syntax Deprecated <pre>SHOW RELATIONSHIP EXISTS CONSTRAINTS</pre>	Replaced by: <pre>SHOW RELATIONSHIP [PROPERTY] EXIST[ENCE] CONSTRAINTS</pre> <p>Still allows BRIEF and VERBOSE but not YIELD or WHERE.</p>
Syntax Deprecated For privilege commands: <pre>ON DEFAULT DATABASE</pre>	Replaced by: <pre>ON HOME DATABASE</pre>
Syntax Deprecated For privilege commands: <pre>ON DEFAULT GRAPH</pre>	Replaced by: <pre>ON HOME GRAPH</pre>
Syntax Deprecated <pre>MATCH (a) RETURN (a)--()</pre>	Pattern expressions producing lists of paths are deprecated, but they can still be used as existence predicates, for example in WHERE clauses. Instead, use a pattern comprehension: <pre>MATCH (a) RETURN [p=(a)--() p]</pre>

Updated features

Feature	Details
Functionality Updated <pre>SHOW INDEXES WHERE ...</pre>	Now allows filtering for: <pre>SHOW INDEXES</pre>
Functionality Updated <pre>SHOW CONSTRAINTS WHERE ...</pre>	Now allows filtering for: <pre>SHOW CONSTRAINTS</pre>

Feature	Details
Functionality Updated	Now allows <code>YIELD</code> , <code>WHERE</code> , and <code>RETURN</code> clauses to <code>SHOW INDEXES</code> to change the output.
SHOW INDEXES <code>YIELD</code> ... [<code>WHERE</code> ...] [<code>RETURN</code> ...]	
Functionality Updated	Now allows <code>YIELD</code> , <code>WHERE</code> , and <code>RETURN</code> clauses to <code>SHOW CONSTRAINTS</code> to change the output.
SHOW CONSTRAINTS <code>YIELD</code> ... [<code>WHERE</code> ...] [<code>RETURN</code> ...]	
Syntax Updated	New syntax for filtering <code>SHOW CONSTRAINTS</code> on property existence constraints. Allows <code>YIELD</code> and <code>WHERE</code> but not <code>BRIEF</code> or <code>VERBOSE</code> .
SHOW [<code>PROPERTY</code>] EXIST[ENCE] CONSTRAINTS	
Syntax Updated	New syntax for filtering <code>SHOW CONSTRAINTS</code> on node property existence constraints. Allows <code>YIELD</code> and <code>WHERE</code> but not <code>BRIEF</code> or <code>VERBOSE</code> .
SHOW NODE [<code>PROPERTY</code>] EXIST[ENCE] CONSTRAINTS	
Syntax Updated	New syntax for filtering <code>SHOW CONSTRAINTS</code> on relationship property existence constraints. Allows <code>YIELD</code> and <code>WHERE</code> but not <code>BRIEF</code> or <code>VERBOSE</code> .
SHOW RELATIONSHIP [<code>PROPERTY</code>] EXIST[ENCE] CONSTRAINTS	
Functionality Updated	Now allows easy filtering for <code>SHOW INDEXES</code> on fulltext indexes. Allows <code>YIELD</code> and <code>WHERE</code> but not <code>BRIEF</code> or <code>VERBOSE</code> .
SHOW FULLTEXT INDEXES	
Functionality Updated	Now allows easy filtering for <code>SHOW INDEXES</code> on token lookup indexes. Allows <code>YIELD</code> and <code>WHERE</code> but not <code>BRIEF</code> or <code>VERBOSE</code> .
SHOW LOOKUP INDEXES	

New features

Feature	Details
Syntax New <pre data-bbox="152 256 382 316"><code>CREATE DATABASE ... [OPTIONS {...}]</code></pre>	New syntax to pass options to CREATE DATABASE . This can be used to specify a specific cluster node to seed data from.
Syntax New <pre data-bbox="152 480 549 563"><code>CREATE CONSTRAINT [name] ON (node:Label) ASSERT node.property IS NOT NULL</code></pre>	New syntax for creating node property existence constraints.
Syntax New <pre data-bbox="152 736 536 819"><code>CREATE CONSTRAINT [name] ON ()-[rel:REL]-() ASSERT rel.property IS NOT NULL</code></pre>	New syntax for creating relationship property existence constraints.
Syntax New <pre data-bbox="152 992 504 1021"><code>ALTER USER name IF EXISTS ...</code></pre>	Makes altering users idempotent. If the specified name does not exists, no error is thrown.
Syntax New <pre data-bbox="152 1208 409 1260"><code>ALTER USER ... SET HOME DATABASE ...</code></pre>	Now allows setting home database for user.
Syntax New <pre data-bbox="152 1432 398 1484"><code>ALTER USER ... REMOVE HOME DATABASE</code></pre>	Now allows removing home database for user.
Syntax New <pre data-bbox="152 1664 409 1715"><code>CREATE USER ... SET HOME DATABASE ...</code></pre>	CREATE USER now allows setting home database for user.
Syntax New <pre data-bbox="152 1882 377 1911"><code>SHOW HOME DATABASE</code></pre>	New syntax for showing the home database of the current user.

Feature	Details
<p>Syntax New</p> <p>New privilege:</p> <pre>SET USER HOME DATABASE</pre>	New Cypher command for administering privilege for changing users home database.
<p>Syntax New</p> <p>For privilege commands:</p> <pre>ON HOME DATABASE</pre>	New syntax for privileges affecting home database.
<p>Syntax New</p> <p>For privilege commands:</p> <pre>ON HOME GRAPH</pre>	New syntax for privileges affecting home graph.
<p>Syntax New</p> <pre>CREATE FULLTEXT INDEX ...</pre>	Allows creating fulltext indexes on nodes or relationships. They can be dropped by using their name.
<p>Functionality New</p> <pre>CREATE INDEX FOR ()-[r:TYPE]-() ...</pre>	Allows creating indexes on relationships with a particular relationship type and property combination. They can be dropped by using their name.
<p>Functionality New</p> <pre>CREATE LOOKUP INDEX ...</pre>	Create token lookup index for nodes with any labels or relationships with any relationship type. They can be dropped by using their name.
<p>Functionality New</p> <pre>RENAME ROLE</pre>	New Cypher command for changing the name of a role.
<p>Functionality New</p> <pre>RENAME USER</pre>	New Cypher command for changing the name of a user.

Feature	Details
<p>Functionality New</p> <pre>SHOW PROCEDURE[S] [EXECUTABLE [BY {CURRENT USER username}]] [YIELD ...] [WHERE ...] [RETURN ...]</pre>	New Cypher commands for listing procedures.
<p>Functionality New</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>Syntax Deprecated</p> <pre>0...</pre>	Replaced by <code>0o...</code>
<p>Syntax Deprecated</p> <pre>0X...</pre>	Only <code>0x...</code> (lowercase x) is supported.
<p>Syntax Deprecated</p> <pre>CALL { RETURN 1 }</pre>	<p>Unaliased expressions are deprecated in subquery <code>RETURN</code> clauses. Replaced by:</p> <pre>CALL { RETURN 1 AS one }</pre>

Updated features

Feature	Details
<p>Functionality Updated</p> <pre>SHOW ROLE name PRIVILEGES</pre>	Can now handle multiple roles.
	<pre>SHOW ROLES n1, n2, ... PRIVILEGES</pre>

Feature	Details
Functionality Updated	Can now handle multiple users.
SHOW USER name PRIVILEGES	SHOW USERS n1, n2, ... PRIVILEGES
Functionality Updated	The <code>round()</code> function can now take an additional argument to specify rounding precision.
<code>round(expression, precision)</code>	
Functionality Updated	The <code>round()</code> function can now take two additional arguments to specify rounding precision and rounding mode.
<code>round(expression, precision, mode)</code>	

New features

Feature	Details
Functionality New	Privileges can now be shown as Cypher commands.
SHOW PRIVILEGES [AS [REVOKE] COMMAND[S]]	
Syntax New	New optional part of the Cypher commands for database privileges .
DEFAULT GRAPH	
Syntax New	Cypher now interprets literals with prefix <code>0o</code> as an octal integer literal.
<code>0o...</code>	
Syntax New	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.
SET [PLAINTEXT ENCRYPTED] PASSWORD	
Functionality New	New Cypher commands for administering privileges for executing procedures and user defined functions. See The DBMS EXECUTE privileges .
New privilege: EXECUTE	

Feature	Details
Syntax New <pre>CREATE [BTREE] INDEX ... [OPTIONS {...}]</pre>	Allows setting index provider and index configuration when creating an index.
Syntax New <pre>CREATE CONSTRAINT ... IS NODE KEY [OPTIONS {...}]</pre>	Allows setting index provider and index configuration for the backing index when creating a node key constraint.
Syntax New <pre>CREATE CONSTRAINT ... IS UNIQUE [OPTIONS {...}]</pre>	Allows setting index provider and index configuration for the backing index when creating a property uniqueness constraint.
Syntax New <pre>SHOW CURRENT USER</pre>	New Cypher command for showing current logged-in user and roles.
Functionality New <pre>SHOW [ALL BTREE] INDEX[ES] [BRIEF VERBOSE [OUTPUT]]</pre>	New Cypher commands for listing indexes. Replaces the procedures <code>db.indexes</code> , <code>db.indexDetails</code> (verbose), and partially <code>db.schemaStatements</code> (verbose).
Functionality New <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. Replaces the procedures <code>db.constraints</code> and partially <code>db.schemaStatements</code> (verbose).
Functionality New New privilege: <pre>SHOW INDEX</pre>	New Cypher command for administering privilege for listing indexes.
Functionality New New privilege: <pre>SHOW CONSTRAINT</pre>	New Cypher command for administering privilege for listing constraints.

Version 4.1.3

New features

Feature	Details
Syntax New <code>CREATE INDEX [name] IF NOT EXISTS FOR ...</code>	Makes index creation idempotent. If an index with the name or schema already exists no error will be thrown.
Syntax New <code>DROP INDEX name IF EXISTS</code>	Makes index deletion idempotent. If no index with the name exists no error will be thrown.
Syntax New <code>CREATE CONSTRAINT [name] IF NOT EXISTS ON ...</code>	Makes constraint creation idempotent. If a constraint with the name or type and schema already exists no error will be thrown.
Syntax New <code>DROP CONSTRAINT name IF EXISTS</code>	Makes constraint deletion idempotent. If no constraint with the name exists no error will be thrown.

Version 4.1

Restricted features

Feature	Details
Functionality Restricted <code>REVOKE ...</code>	No longer revokes sub-privileges when revoking a compound privilege, e.g. when revoking INDEX MANAGEMENT , any CREATE INDEX and DROP INDEX privileges will no longer be revoked.
Functionality Restricted <code>ALL DATABASE PRIVILEGES</code>	No longer includes the privileges START DATABASE and STOP DATABASE .

Updated features

Feature	Details
Procedure Updated queryId	The <code>queryId</code> procedure format has changed, and no longer includes the database name. For example, <code>mydb-query-123</code> is now <code>query-123</code> . This change affects built-in procedures <code>dbms.listQueries()</code> , <code>dbms.listActiveLocks(queryId)</code> , <code>dbms.killQueries(queryIds)</code> and <code>dbms.killQuery(queryId)</code> .
Functionality Updated SHOW PRIVILEGES	The returned privileges are a closer match to the original grants and denies, e.g. if granted <code>MATCH</code> the command will show that specific privilege and not the <code>TRAVERSE</code> and <code>READ</code> privileges. Added support for <code>YIELD</code> and <code>WHERE</code> clauses to allow filtering results.

New features

Feature	Details
Functionality New New role: PUBLIC	The <code>PUBLIC</code> role is automatically assigned to all users, giving them a set of base privileges.
Syntax New For privileges: REVOKE MATCH	The <code>MATCH</code> privilege can now be revoked.
Functionality New SHOW USERS	New support for <code>YIELD</code> and <code>WHERE</code> clauses to allow filtering results.
Functionality New SHOW ROLES	New support for <code>YIELD</code> and <code>WHERE</code> clauses to allow filtering results.
Functionality New SHOW DATABASES	New support for <code>YIELD</code> and <code>WHERE</code> clauses to allow filtering results.

Feature	Details
Functionality New TRANSACTION MANAGEMENT privileges	New Cypher commands for administering transaction management.
Functionality New DBMS USER MANAGEMENT privileges	New Cypher commands for administering user management.
Functionality New DBMS DATABASE MANAGEMENT privileges	New Cypher commands for administering database management.
Functionality New DBMS PRIVILEGE MANAGEMENT privileges	New Cypher commands for administering privilege management.
Functionality New ALL DBMS PRIVILEGES	New Cypher command for administering role, user, database and privilege management.
Functionality New ALL GRAPH PRIVILEGES	New Cypher command for administering read and write privileges.
Functionality New Write privileges	New Cypher commands for administering write privileges.
Functionality New ON DEFAULT DATABASE	New optional part of the Cypher commands for database privileges.

Version 4.0

Removed features

Feature	Details
Function Removed rels()	Replaced by relationships() .

Feature	Details
<p>Function Removed</p> <p><code>toInt()</code></p>	Replaced by toInteger() .
<p>Function Removed</p> <p><code>lower()</code></p>	Replaced by toLower() .
<p>Function Removed</p> <p><code>upper()</code></p>	Replaced by toUpper() .
<p>Function Removed</p> <p><code>extract()</code></p>	Replaced by list comprehension .
<p>Function Removed</p> <p><code>filter()</code></p>	Replaced by list comprehension .
<p>Functionality Removed</p> <p>For Rule planner:</p> <p><code>CYPHER planner=rule</code></p>	The RULE planner was removed in 3.2, but still possible to trigger using START or CREATE UNIQUE clauses. Now it is completely removed.
<p>Functionality Removed</p> <p>Explicit indexes</p>	The removal of the RULE planner in 3.2 was the beginning of the end for explicit indexes. Now they are completely removed, including the removal of the built-in procedures for Neo4j 3.3 to 3.5 .
<p>Functionality Removed</p> <p>For compiled runtime:</p> <p><code>CYPHER runtime=compiled</code></p>	Replaced by the new pipelined runtime which covers a much wider range of queries.
<p>Clause Removed</p> <p><code>CREATE UNIQUE</code></p>	Running queries with this clause will cause a syntax error.

Feature	Details
Clause Removed	Running queries with this clause will cause a syntax error.
START	
Syntax Removed	Replaced by <code>MATCH (n)-[:A B C {foo: 'bar'}]-()</code> <code>RETURN n</code> .
<code>MATCH (n)-[:A B C {foo: 'bar'}]-() RETURN n</code>	
Syntax Removed	Replaced by <code>MATCH (n)-[x:A B C]-()</code> <code>RETURN n</code> .
<code>MATCH (n)-[x:A B C]-() RETURN n</code>	
Syntax Removed	Replaced by <code>MATCH (n)-[x:A B C*]-()</code> <code>RETURN n</code> .
<code>MATCH (n)-[x:A B C*]-() RETURN n</code>	
Syntax Removed	Replaced by <code>\$parameter</code> .
<code>{parameter}</code>	

Deprecated features

Feature	Details
Syntax Deprecated	As in Cypher 3.2, this is replaced by:
<code>MATCH (n)-[rs*]-() RETURN rs</code>	<code>MATCH p=(n)-[*]-() RETURN relationships(p) AS rs</code>
Syntax Deprecated	Replaced by <code>CREATE INDEX FOR (n:Label) ON (n.prop)</code> .
<code>CREATE INDEX ON :Label(prop)</code>	
Syntax Deprecated	Replaced by <code>DROP INDEX name</code> .
<code>DROP INDEX ON :Label(prop)</code>	

Feature	Details
Syntax <code>DROP CONSTRAINT ON (n:Label) ASSERT (n.prop) IS NODE KEY</code>	Deprecated Replaced by <code>DROP CONSTRAINT name</code> .
Syntax <code>DROP CONSTRAINT ON (n:Label) ASSERT (n.prop) IS UNIQUE</code>	Deprecated Replaced by <code>DROP CONSTRAINT name</code> .
Syntax <code>DROP CONSTRAINT ON (n:Label) ASSERT exists(n.prop)</code>	Deprecated Replaced by <code>DROP CONSTRAINT name</code> .
Syntax <code>DROP CONSTRAINT ON ()-[r:Type]-() ASSERT exists (r.prop)</code>	Deprecated Replaced by <code>DROP CONSTRAINT name</code> .

Restricted features

Feature	Details
Function <code>length()</code>	Restricted to only work on paths. See <code>length()</code> for more details.
Function <code>size()</code>	No longer works for paths. Only works for strings, lists and pattern expressions. See <code>size()</code> for more details.

Updated features

Feature	Details
<p>Syntax Extended</p> <pre>CREATE CONSTRAINT [name] ON ...</pre>	<p>The create constraint syntax can now include a name.</p> <p>The <code>IS NODE KEY</code> and <code>IS UNIQUE</code> versions of this command replace the procedures <code>db.createNodeKey</code> and <code>db.createUniquePropertyConstraint</code>, respectively.</p>

New features

Feature	Details
<p>Functionality New</p> <p>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>Functionality New</p> <p>Multi-database administration</p>	<p>New Cypher commands for administering multiple databases.</p>
<p>Functionality New</p> <p>Access control</p>	<p>New Cypher commands for administering role-based access control.</p>
<p>Functionality New</p> <p>Fine-grained security</p>	<p>New Cypher commands for administering dbms, database, graph and sub-graph access control.</p>
<p>Syntax New</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>Replaces the <code>db.createIndex</code> procedure.</p>
<p>Syntax New</p> <pre>DROP INDEX name</pre>	<p>New command for dropping an index by name.</p>
<p>Syntax New</p> <pre>DROP CONSTRAINT name</pre>	<p>New command for dropping a constraint by name, no matter the type.</p>

Feature	Details
<p>Clause New</p> <pre>WHERE EXISTS {...}</pre>	EXISTS subqueries are subclauses used to filter the results of a MATCH , OPTIONAL MATCH , or WITH clause.
<p>Clause New</p> <pre>USE neo4j</pre>	New clause to specify which graph a query, or query part, is executed against.

Version 3.5

Deprecated features

Feature	Details
<p>Functionality Deprecated</p> <p>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>Function Deprecated</p> <pre>extract()</pre>	Replaced by list comprehension .
<p>Function Deprecated</p> <pre>filter()</pre>	Replaced by list comprehension .

Version 3.4

Feature	Type	Change	Details
Spatial point types	Functionality	Amendment	A point — irrespective of which Coordinate Reference System is used — can be stored as a property and is able to be backed by an index. Prior to this, a point was a virtual property only.
point() - Cartesian 3D	Function	Added	
point() - WGS 84 3D	Function	Added	

Feature	Type	Change	Details
randomUUID()	Function	Added	
Temporal types	Functionality	Added	Supports storing, indexing and working with the following temporal types: Date, Time, LocalTime, DateTime, LocalDateTime and Duration.
Temporal functions	Functionality	Added	Functions allowing for the creation and manipulation of values for each temporal type — Date, Time, LocalTime, DateTime, LocalDateTime and Duration.
Temporal operators	Functionality	Added	Operators allowing for the manipulation of values for each temporal type — Date, Time, LocalTime, DateTime, LocalDateTime and Duration.
toString()	Function	Extended	Now also allows temporal values as input (i.e. values of type Date, Time, LocalTime, DateTime, LocalDateTime or Duration).

Version 3.3

Feature	Type	Change	Details
<code>START</code>	Clause	Removed	As in Cypher 3.2, any queries using the <code>START</code> clause will revert back to Cypher 3.1 <code>planner=rule</code> . However, there are built-in procedures for Neo4j versions 3.3 to 3.5 for accessing explicit indexes. The procedures will enable users to use the current version of Cypher and the cost planner together with these indexes. An example of this is <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

Feature	Type	Change	Details
<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.
<code>CREATE UNIQUE</code>	Clause	Removed	Running such queries will fall back to using Cypher 3.1 (and use the rule planner)
<code>START</code>	Clause	Removed	Running such queries will fall back to using Cypher 3.1 (and use the rule planner)
<code>MATCH (n)-[rs*]-() RETURN rs</code>	Syntax	Deprecated	Replaced by <code>MATCH p=(n)-[*]-() RETURN relationships(p) AS rs</code>
<code>MATCH (n)-[:A B C {foo: 'bar'}]-() RETURN n</code>	Syntax	Deprecated	Replaced by <code>MATCH (n)-[:A B C {foo: 'bar'}]-() RETURN n</code>
<code>MATCH (n)-[x:A B C]-() RETURN n</code>	Syntax	Deprecated	Replaced by <code>MATCH (n)-[x:A B C]-() RETURN n</code>
<code>MATCH (n)-[x:A B C*]-() RETURN n</code>	Syntax	Deprecated	Replaced by <code>MATCH (n)-[x:A B C*]-() RETURN n</code>
User-defined aggregation functions	Functionality	Added	
Composite indexes	Index	Added	
Node Key	Index	Added	Neo4j Enterprise Edition only
<code>CYPHER runtime=compiled</code> (Compiled runtime)	Functionality	Added	Neo4j Enterprise Edition only
<code>reverse()</code>	Function	Extended	Now also allows a list as input
<code>max()</code> , <code>min()</code>	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 relationships()
<code>toInt()</code>	Function	Deprecated	Replaced by toInteger()
<code>lower()</code>	Function	Deprecated	Replaced by toLower()
<code>upper()</code>	Function	Deprecated	Replaced by toUpper()
<code>toBoolean()</code>	Function	Added	
Map projection	Syntax	Added	
Pattern comprehension	Syntax	Added	
User-defined functions	Functionality	Added	
CALL...YIELD...WHERE	Clause	Extended	Records returned by YIELD may be filtered further using WHERE

Version 3.0

Feature	Type	Change	Details
<code>has()</code>	Function	Removed	Replaced by exists()
<code>str()</code>	Function	Removed	Replaced by toString()
{parameter}	Syntax	Deprecated	Replaced by \$parameter
<code>properties()</code>	Function	Added	
CALL [...YIELD]	Clause	Added	
<code>point() - Cartesian 2D</code>	Function	Added	
<code>point() - WGS 84 2D</code>	Function	Added	
<code>distance()</code>	Function	Added	
User-defined procedures	Functionality	Added	
<code>toString()</code>	Function	Extended	Now also allows Boolean values as input

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
<code>CALL [...YIELD]</code>	Reading/Writing	Invoke a procedure deployed in the database.
<code>CALL {...}</code>	Reading/Writing	Evaluates a subquery, typically used for post-union processing or aggregations.
<code>CALL { ... } IN TRANSACTIONS</code>	Reading/Writing	Evaluates a subquery in separate transactions. Typically used when modifying or importing large amounts of data.
<code>CREATE</code>	Writing	Create nodes and relationships.
<code>CREATE CONSTRAINT [existence] [IF NOT EXISTS] FOR (n:Label) REQUIRE n.property IS NOT NULL [OPTIONS {}]</code>	Schema	Create a constraint ensuring that all nodes with a particular label have a certain property.
<code>CREATE CONSTRAINT [existence] [IF NOT EXISTS] FOR ()-["r:REL_TYPE"]->() REQUIRE r.property IS NOT NULL [OPTIONS {}]</code>	Schema	Create a constraint that ensures all relationships with a particular type have a certain property.
<code>CREATE CONSTRAINT [node_key] [IF NOT EXISTS] FOR (n:Label) REQUIRE (n.prop1[, ..., n.propN]) IS [NODE] KEY [OPTIONS {optionKey: optionValue[, ...]}]</code>	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.

Clause	Category	Description
<code>CREATE CONSTRAINT [uniqueness] [IF NOT EXISTS] FOR (n:Label) REQUIRE (n.prop1[, ..., n.propN]) IS [NODE] UNIQUE [OPTIONS {optionKey: optionValue[, ...]}]]</code>	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.
<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 POINT INDEX [name] [IF NOT EXISTS] FOR (n:Label) ON (n.property) [OPTIONS {optionKey: optionValue[, ...]}]]</code>	Schema	Create a point index on nodes.
<code>CREATE POINT INDEX [name] [IF NOT EXISTS] FOR ()-["r:TYPE" "-() ON (r.property) [OPTIONS {optionKey: optionValue[, ...]}]]</code>	Schema	Create a point index on relationships.
<code>CREATE [RANGE] INDEX [name] [IF NOT EXISTS] FOR (n:Label) ON (n.property[, ..., n.propertyN]) [OPTIONS {optionKey: optionValue[, ...]}]]</code>	Schema	Create a range index on nodes.
<code>CREATE [RANGE] INDEX [name] [IF NOT EXISTS] FOR ()-["r:TYPE" "-() ON (r.property[, ..., r.propertyN]) [OPTIONS {optionKey: optionValue[, ...]}]]</code>	Schema	Create a range index on relationships.
<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.

Clause	Category	Description
<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.
<code>DETACH DELETE</code>	Writing	Delete a node or set of nodes. All associated relationships will automatically be deleted.
<code>DROP CONSTRAINT name [IF EXISTS]</code>	Schema	Drop a constraint using the name.
<code>DROP INDEX name [IF EXISTS]</code>	Schema	Drop an index using the name.
<code>FOREACH</code>	Writing	Update data within a list, whether components of a path, or the result of aggregation.
<code>LIMIT</code>	Reading sub-clause	A sub-clause used to constrain the number of rows in the output.
<code>LOAD CSV</code>	Importing data	Use when importing data from CSV files.
<code>MATCH</code>	Reading	Specify the patterns to search for in the database.
<code>MERGE</code>	Reading/Writing	Ensures that a pattern exists in the graph. Either the pattern already exists, or it needs to be created.
<code>ON CREATE</code>	Reading/Writing	Used in conjunction with <code>MERGE</code> , specifying the actions to take if the pattern needs to be created.
<code>ON MATCH</code>	Reading/Writing	Used in conjunction with <code>MERGE</code> , specifying the actions to take if the pattern already exists.
<code>OPTIONAL MATCH</code>	Reading	Specify the patterns to search for in the database while using <code>nulls</code> for missing parts of the pattern.
<code>ORDER BY [ASC[ENDING] DESC[ENDING]\]</code>	Reading sub-clause	A sub-clause following <code>RETURN</code> or <code>WITH</code> , specifying that the output should be sorted in either ascending (the default) or descending order.
<code>REMOVE</code>	Writing	Remove properties and labels from nodes and relationships.
<code>RETURN ... [AS]</code>	Projecting	Defines what to include in the query result set.

Clause	Category	Description
SET	Writing	Update labels on nodes and properties on nodes and relationships.
SHOW [ALL UNIQUENESS]NODE [PROPERTY] EXIST[ENCE] REL[ATIONSHIP] [PROPERTY] EXIST[ENCE] [[PROPERTY]] EXIST[ENCE] NODE KEY] CONSTRAINT[S]]	Schema	List constraints in the database, either all or filtered on type. Also allows WHERE and YIELD clauses.
SHOW [ALL FULLTEXT LOOKUP POINT RANGE TEXT] INDEX[ES]]	Schema	List indexes in the database, either all or filtered on fulltext, lookup, point, range, or text indexes. Also allows WHERE and YIELD clauses.
SHOW [ALL BUILT IN USER DEFINED] FUNCTION[S] [EXECUTABLE [BY {CURRENT USER username}]\]]	DBMS	List functions, either all or filtered. Available filters are executable by a user or function type (built-in or user-defined). Also allows WHERE and YIELD clauses.
SHOW PROCEDURE[S] [EXECUTABLE [BY {CURRENT USER username}]\]]	DBMS	List procedures, either all or filtered on executable by a user. Also allows WHERE and YIELD clauses.
SHOW TRANSACTION[S] [transaction-id[, ...]\]]	DBMS	List transactions, either all or filtered on ID. Also allows WHERE and YIELD clauses.
SKIP	Reading/Writing	A sub-clause defining from which row to start including the rows in the output.
TERMINATE TRANSACTION[S] transaction-id[, ...]\]]	DBMS	Terminate transactions with the given IDs.
UNION	Set operations	Combines the result of multiple queries. Duplicates are removed.
UNION ALL	Set operations	Combines the result of multiple queries. Duplicates are retained.
UNWIND ... [AS]	Projecting	Expands a list into a sequence of rows.
USE	Multiple graphs	Determines which graph a query, or query part, is executed against.
USING INDEX variable:Label(property)	Hint	Index hints are used to specify which index, if any, the planner should use as a starting point.
USING INDEX SEEK variable:Label(property)	Hint	Index seek hint instructs the planner to use an index seek for this clause.
USING JOIN ON variable	Hint	Join hints are used to enforce a join operation at specified points.

Clause	Category	Description
USING SCAN variable:Label	Hint	Scan hints are used to force the planner to do a label scan (followed by a filtering operation) instead of using an index.
WITH ... [AS]	Projecting	Allows query parts to be chained together, piping the results from one to be used as starting points or criteria in the next.
WHERE	Reading sub-clause	A sub-clause used to add constraints to the patterns in a MATCH or OPTIONAL MATCH clause, or to filter the results of a WITH 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.

Operator	Category	Description
>	Comparison	Greater than.
<<query-operators-comparison, >=>	Comparison	Greater than or equal to.
AND	Boolean	Conjunction.
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.
[N]	Map	Subscript (dynamic value access by key).
[N]	Property	Subscript (dynamic property access).
[N]	List	Subscript (accessing element(s) in a list).
^	Mathematical	Exponentiation.

Functions

Function	Category	Description
<code>abs()</code>	Numeric	Returns the absolute value of a number.
<code>acos()</code>	Trigonometric	Returns the arccosine of a number in radians.
<code>all()</code>	Predicate	Tests whether the predicate holds for all elements in a list.
<code>any()</code>	Predicate	Tests whether the predicate holds for at least one element in a list.
<code>asin()</code>	Trigonometric	Returns the arcsine of a number in radians.
<code>atan()</code>	Trigonometric	Returns the arctangent of a number in radians.
<code>atan2()</code>	Trigonometric	Returns the arctangent2 of a set of coordinates in radians.
<code>avg()</code>	Aggregating	Returns the average of a set of values.

Function	Category	Description
<code>ceil()</code>	Numeric	Returns the smallest floating point number that is greater than or equal to a number and equal to a mathematical integer.
<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. Truncation summary .
<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.

Function	Category	Description
<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.
<code>datetime({epochSeconds})</code>	Temporal	Returns a DateTime from a timestamp.
<code>datetime.realpath()</code>	Temporal	Returns the current DateTime using the <code>realtime</code> clock.
<code>datetime.statement()</code>	Temporal	Returns the current DateTime using the <code>statement</code> clock.
<code>datetime.transaction()</code>	Temporal	Returns the current DateTime using the <code>transaction</code> clock.
<code>datetime.truncate()</code>	Temporal	Returns a DateTime obtained by truncating a value at a specific component boundary. Truncation summary .
<code>degrees()</code>	Trigonometric	Converts radians to degrees.
<code>duration({map})</code>	Temporal	Returns a Duration from a map of its components.
<code>duration(string)</code>	Temporal	Returns a Duration by parsing a string.
<code>duration.between()</code>	Temporal	Returns a Duration equal to the difference between two given instants.
<code>duration.inDays()</code>	Temporal	Returns a Duration equal to the difference in whole days or weeks between two given instants.
<code>duration.inMonths()</code>	Temporal	Returns a Duration equal to the difference in whole months, quarters or years between two given instants.
<code>duration.inSeconds()</code>	Temporal	Returns a Duration 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, <code>e</code> .
<code>endNode()</code>	Scalar	Returns the end node of a relationship.
<code>exists()</code>	Predicate	Returns <code>true</code> if a match for the pattern exists in the graph.
<code>exp()</code>	Logarithmic	Returns e^n , where <code>e</code> is the base of the natural logarithm, and <code>n</code> is the value of the argument expression.

Function	Category	Description
<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.
<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>isNaN()</code>	Numeric	Returns <code>true</code> if the given numeric value is <code>NaN</code> (Not a Number).
<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.

Function	Category	Description
<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. Truncation summary .
<code>localtime()</code>	Temporal	Returns the current <code>LocalTime</code> .
<code>localtime({hour [, minute, second, ...]})</code>	Temporal	Returns a <code>LocalTime</code> with the specified component values.
<code>localtime(string)</code>	Temporal	Returns a <code>LocalTime</code> by parsing a string.
<code>localtime({time [, hour, ...]})</code>	Temporal	Returns a <code>LocalTime</code> from a map of another temporal value's components.
<code>localtime.realpath()</code>	Temporal	Returns the current <code>LocalTime</code> using the <code>realtime</code> clock.
<code>localtime.statement()</code>	Temporal	Returns the current <code>LocalTime</code> using the <code>statement</code> clock.
<code>localtime.transaction()</code>	Temporal	Returns the current <code>LocalTime</code> using the <code>transaction</code> clock.
<code>localtime.truncate()</code>	Temporal	Returns a <code>LocalTime</code> obtained by truncating a value at a specific component boundary. Truncation summary .
<code>log()</code>	Logarithmic	Returns the natural logarithm of a number.
<code>log10()</code>	Logarithmic	Returns the common logarithm (base 10) of a number.
<code>lTrim()</code>	String	Returns the original string with leading whitespace removed.
<code>max()</code>	Aggregating	Returns the maximum value in a set of values.
<code>min()</code>	Aggregating	Returns the minimum value in a set of values.
<code>nodes()</code>	List	Returns a list containing all the nodes in a path.
<code>none()</code>	Predicate	Returns true if the predicate holds for no element in a list.
<code>percentileCont()</code>	Aggregating	Returns the percentile of the given value over a group using linear interpolation.
<code>percentileDisc()</code>	Aggregating	Returns the nearest value to the given percentile over a group using a rounding method.

Function	Category	Description
pi()	Trigonometric	Returns the mathematical constant pi.
point() - Cartesian 2D	Spatial	Returns a 2D point object, given two coordinate values in the Cartesian coordinate system.
point() - Cartesian 3D	Spatial	Returns a 3D point object, given three coordinate values in the Cartesian coordinate system.
point() - WGS 84 2D	Spatial	Returns a 2D point object, given two coordinate values in the WGS 84 coordinate system.
point() - WGS 84 3D	Spatial	Returns a 3D point object, given three coordinate values in the WGS 84 coordinate system.
point.distance()	Spatial	Returns true if the provided point is within the bounding box defined by the two provided points.
point.withinBBox()	Spatial	Returns a floating point number representing the geodesic distance between any two points in the same CRS.
properties()	Scalar	Returns a map containing all the properties of a node or relationship.
radians()	Trigonometric	Converts degrees to radians.
rand()	Numeric	Returns a random floating point number in the range from 0 (inclusive) to 1 (exclusive); i.e. [0, 1).
randomUUID()	Scalar	Returns a string value corresponding to a randomly-generated UUID.
range()	List	Returns a list comprising all integer values within a specified range.
reduce()	List	Runs an expression against individual elements of a list, storing the result of the expression in an accumulator.
relationships()	List	Returns a list containing all the relationships in a path.
replace()	String	Returns a string in which all occurrences of a specified string in the original string have been replaced by another (specified) string.
reverse()	List	Returns a list in which the order of all elements in the original list have been reversed.

Function	Category	Description
reverse()	String	Returns a string in which the order of all characters in the original string have been reversed.
right()	String	Returns a string containing the specified number of rightmost characters of the original string.
round()	Numeric	Returns the floating point value of the given number rounded to the nearest mathematical integer, with half-way values always rounded up.
round(), with precision	Numeric	Returns the floating point value of the given number rounded with the specified precision, with half-values always being rounded up.
round(), with precision and rounding mode	Numeric	Returns the floating point value of the given number rounded with the specified precision and the specified rounding mode.
rTrim()	String	Returns the original string with trailing whitespace removed.
sign()	Numeric	Returns the signum of a number: 0 if the number is 0 , -1 for any negative number, and 1 for any positive number.
sin()	Trigonometric	Returns the sine of a number.
single()	Predicate	Returns true if the predicate holds for exactly one of the elements in a list.
size()	Scalar	Returns the number of items in a list.
size() applied to pattern comprehension	Scalar	Returns the number of paths matching the pattern comprehension.
size() applied to string	Scalar	Returns the number of Unicode characters in a string.
split()	String	Returns a list of strings resulting from the splitting of the original string around matches of the given delimiter.
sqrt()	Logarithmic	Returns the square root of a number.
startNode()	Scalar	Returns the start node of a relationship.
stDev()	Aggregating	Returns the standard deviation for the given value over a group for a sample of a population.
stDevP()	Aggregating	Returns the standard deviation for the given value over a group for an entire population.

Function	Category	Description
<code>substring()</code>	String	Returns a substring of the original string, beginning with a 0-based index start and length.
<code>sum()</code>	Aggregating	Returns the sum of a set of numeric values.
<code>tail()</code>	List	Returns all but the first element in a list.
<code>tan()</code>	Trigonometric	Returns the tangent of a number.
<code>time()</code>	Temporal	Returns the current Time.
<code>time({hour [, minute, ...]})</code>	Temporal	Returns a Time with the specified component values.
<code>time(string)</code>	Temporal	Returns a Time by parsing a string.
<code>time({time [, hour, ..., timezone]})</code>	Temporal	Returns a Time from a map of another temporal value's components.
<code>time.realtime()</code>	Temporal	Returns the current Time using the <code>realtime</code> clock.
<code>time.statement()</code>	Temporal	Returns the current Time using the <code>statement</code> clock.
<code>time.transaction()</code>	Temporal	Returns the current Time using the <code>transaction</code> clock.
<code>time.truncate()</code>	Temporal	Returns a Time obtained by truncating a value at a specific component boundary. Truncation summary .
<code>timestamp()</code>	Scalar	Returns the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC.
<code>toBoolean()</code>	Scalar	Converts a string value to a boolean value.
<code>toFloat()</code>	Scalar	Converts an integer or string value to a floating point number.
<code>toInteger()</code>	Scalar	Converts a floating point or string value to an integer value.
<code>toLower()</code>	String	Returns the original string in lowercase.
<code>toString()</code>	String	Converts an integer, float, boolean or temporal (i.e. Date, Time, LocalTime, DateTime, LocalDateTime or Duration) value to a string.
<code>toUpper()</code>	String	Returns the original string in uppercase.
<code>trim()</code>	String	Returns the original string with leading and trailing whitespace removed.

Function	Category	Description
<code>type()</code>	Scalar	Returns the string representation of the relationship type.

Expressions

Name	Description
<code>CASE Expression</code>	A generic conditional expression, similar to if/else statements available in other languages.
<code>EXISTS {...}</code>	An <code>EXISTS</code> expression is used to evaluate the existence of a subquery.
<code>COUNT {...}</code>	An expression used to compute the number of results of a subquery.

Cypher query options

Name	Type	Description //// Removed in 5.0
<code>CYPHER \$version query</code>	Version	This will force ' <code>query</code> ' to use Neo4j Cypher <code>\$version</code> . ////
<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 ' <code>query</code> '. 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.

Command	Admin category	Description
<code>ALTER DATABASE ... [IF EXISTS] [SET ACCESS {READ ONLY READ WRITE}] [SET TOPOLOGY n PRIMAR{Y IES} [m SECONDAR{Y IES}]]]</code>	Database	Modifies the database access mode and / or topology.
<code>ALTER SERVER ... [SET OPTIONS\ {...}]</code>	Server management	Modifies the options for a server.
<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] COMPOSITE DATABASE ... [IF NOT EXISTS] [OPTIONS {}] [WAIT [n [SECONDON[S]]]] NOWAIT]]</code>	Database	Creates a new composite database.
<code>CREATE [OR REPLACE] DATABASE ... [IF NOT EXISTS] [TOPOLOGY n PRIMAR{Y IES} [m SECONDAR{Y IES}]] [OPTIONS {optionKey: optionValue[, ...]}] [WAIT [n [SECONDON[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.
<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>DEALLOCATE DATABASE(S) FROM SERVER(S) ...</code>	Server management	Removes databases from the specified servers.
<code>DENY [IMMUTABLE] ... ON DATABASE ... TO]</code>	Privilege	Denies a database or schema privilege to one or multiple roles.
<code>DENY [IMMUTABLE] ... ON DBMS TO]</code>	Privilege	Denies a DBMS privilege to one or multiple roles.
<code>DENY [IMMUTABLE] ... ON GRAPH ... [NODES RELATIONSHIPS ELEMENTS] ... TO]</code>	Privilege	Denies a graph privilege for one or multiple specified elements to one or multiple roles.

Command	Admin category	Description
<code>DROP ALIAS ... [IF EXISTS] FOR DATABASE]</code>	Database alias	Deletes a specified database alias.
<code>DROP COMPOSITE DATABASE ... [IF EXISTS] [DUMP DATA DESTROY DATA] [WAIT [n [SECOND[S]]]]] [NOWAIT]]</code>	Database	Deletes a specified composite database.
<code>DROP DATABASE ... [IF EXISTS] [DUMP DATA DESTROY DATA]]</code>	Database	Deletes a specified database (either standard or composite).
<code>DROP ROLE ... [IF EXISTS]</code>	User and role	Deletes a specified role.
<code>DROP SERVER ...</code>	Server management	Removes a specified server.
<code>DROP USER ... [IF EXISTS]</code>	User and role	Deletes a specified user.
<code>ENABLE SERVER [OPTIONS]</code>	Server management	Enables a specified server.
<code>GRANT [IMMUTABLE] ... ON DATABASE ... TO]</code>	Privilege	Assigns a database or schema privilege to one or multiple roles.
<code>GRANT [IMMUTABLE] ... ON DBMS TO]</code>	Privilege	Assigns a DBMS privilege to one or multiple roles.
<code>GRANT [IMMUTABLE] ... 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 [IMMUTABLE] ROLE[S] ... TO]</code>	User and role	Assigns one or multiple roles to one or multiple users.
<code>REALLOCATE DATABASE(S)</code>	Server management	Re-balance databases among all servers.
<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 [IMMUTABLE] [GRANT DENY] ... ON DATABASE ... FROM]</code>	Privilege	Removes a database or schema privilege from one or multiple roles.
<code>REVOKE [IMMUTABLE] [GRANT DENY] ... ON DBMS FROM]</code>	Privilege	Removes a DBMS privilege from one or multiple roles.
<code>REVOKE [IMMUTABLE] [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.
<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.

Command	Admin category	Description
SHOW DATABASE	Database	Returns information about a specified database.
SHOW DATABASES	Database	Returns information about all databases.
SHOW SERVERS	Server management	Returns information about all servers.
SHOW DEFAULT DATABASE	Database	Returns information about the default database.
SHOW HOME DATABASE	Database	Returns information about the current users home database.
SHOW [ROLE ... USER ... ALL \ PRIVILEGES [AS [REVOKE]\ COMMAND[S]\]]	Privilege	Returns information about role, user or all privileges.
SHOW USERS	User and role	Returns information about all users.
START DATABASE	Database	Starts up a specified database.
STOP DATABASE	Database	Stops a specified database.

Privilege Actions

Name	Category	Description
ACCESS	Database	Determines whether a user can access a specific database.
ALL DATABASE PRIVILEGES	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.
ALL DBMS PRIVILEGES	DBMS	Determines whether a user is allowed to perform role, user, database and privilege management.
ALL GRAPH PRIVILEGES	GRAPH	Determines whether a user is allowed to perform reads and writes.
ALTER ALIAS	DBMS	Determines whether the user can modify aliases.
ALTER DATABASE	DBMS	Determines whether the user can modify databases and aliases.
ALTER USER	DBMS	Determines whether the user can modify users.
ASSIGN PRIVILEGE	DBMS	Determines whether the user can assign privileges using the GRANT and DENY commands.
ASSIGN ROLE	DBMS	Determines whether the user can grant roles.

Name	Category	Description
COMPOSITE DATABASE MANAGEMENT	DBMS	Determines whether the user can create and delete composite databases.
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 COMPOSITE DATABASE	DBMS	Determines whether the user can create new composite databases.
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 COMPOSITE DATABASE	DBMS	Determines whether the user can delete composite databases.
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 gets elevated privileges when executing functions.
EXECUTE BOOSTED PROCEDURE	DBMS	Determines whether the user gets elevated privileges when executing procedures.
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 <code>REMOVE</code> clause.
REMOVE PRIVILEGE	DBMS	Determines whether the user can remove privileges using the <code>REVOKE</code> 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.
SERVER MANAGEMENT	DBMS	Determines whether the user can enable, alter, rename, reallocate, deallocate, drop, and show servers.
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 <code>SET</code> 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 <code>SET</code> 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.

Name	Category	Description
SHOW INDEX	Schema	Determines whether the user is allowed to list indexes.
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 SERVERS	DBMS	Determines whether the user can get information about servers.
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

```
CREATE (a:End {prop: 42}),
       (b:End {prop: 3}),
       (c:Begin {prop: elementId(a)})
```

Good

```
CREATE (a:End {prop: 42}),
       (:End {prop: 3}),
       (:Begin {prop: elementId(a)})
```

- Chain patterns together to avoid repeating variables.

Bad

```
MATCH (:Person)-->(:vehicle:Car), (:vehicle:Car)-->(:Company)
RETURN count(vehicle)
```

Good

```
MATCH (:Person)-->(:vehicle:Car)-->(:Company)
RETURN count(vehicle)
```

- Put named nodes before anonymous nodes.

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>.