The Neo4j Cypher Manual v4.0
Table of Contents

1. Introduction .................................................................................................................. 2
   1.1. What is Cypher? ................................................................................................. 2
   1.2. Neo4j databases and graphs ............................................................................. 4
   1.3. Querying, updating and administering ............................................................. 5
   1.4. Transactions ....................................................................................................... 7
   1.5. Cypher path matching ....................................................................................... 8
2. Syntax ............................................................................................................................ 12
   2.1. Values and types .............................................................................................. 14
   2.2. Naming rules and recommendations .................................................................. 16
   2.3. Expressions ....................................................................................................... 18
   2.4. Variables ........................................................................................................... 23
   2.5. Reserved keywords ........................................................................................... 24
   2.6. Parameters ........................................................................................................ 27
   2.7. Operators ......................................................................................................... 31
3. Using a regular expression with =~ to filter words ...................................................... 41
4. Boolean operators ....................................................................................................... 42
   4.1. Using boolean operators to filter numbers ...................................................... 42
5. String operators .......................................................................................................... 43
   5.1. Concatenating two strings with + .................................................................... 43
6. Temporal operators ..................................................................................................... 44
   6.1. Adding and subtracting a Duration to or from a temporal instant .................... 44
   6.2. Adding and subtracting a Duration to or from another Duration .................... 45
   6.3. Multiplying and dividing a Duration with or by a number ............................... 46
7. Map operators .............................................................................................................. 47
   7.1. Statically accessing the value of a nested map by key using the . operator ...... 47
   7.2. Dynamically accessing the value of a map by key using the [] operator and a parameter .... 47
8. List operators .............................................................................................................. 48
   8.1. Concatenating two lists using + ..................................................................... 48
   8.2. Using IN to check if a number is in a list ........................................................ 48
   8.3. Using IN for more complex list membership operations .................................. 48
   8.4. Accessing elements in a list using the [] operator ........................................... 49
   8.5. Dynamically accessing an element in a list using the [] operator and a parameter . . . 50
   8.6. Using IN with [] on a nested list .................................................................... 50
   8.7. Comments ........................................................................................................ 51
   8.8. Patterns ............................................................................................................ 51
   8.9. Temporal (Date/Time) values ......................................................................... 56
   8.10. Spatial values .................................................................................................. 74
   8.11. Lists ............................................................................................................... 81
This is the Cypher manual for Neo4j version 4.0, authored by the Neo4j Team.

This manual covers the following areas:

- **Introduction** — Introducing the Cypher query language.
- **Syntax** — Learn Cypher query syntax.
- **Clauses** — Reference of Cypher query clauses.
- **Functions** — Reference of Cypher query functions.
- **Administration** — Working with databases, indexes, constraints and security in Cypher.
- **Query tuning** — Learn to analyze queries and tune them for performance.
- **Execution plans** — Cypher execution plans and operators.
- **Deprecations, additions and compatibility** — An overview of language developments across versions.
- **Glossary of keywords** — A glossary of Cypher keywords, with links to other parts of the Cypher manual.
- **Cypher styleguide** — A guide to the recommended style for writing Cypher queries.

Who should read this?

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

This section provides an introduction to the Cypher query language.

1.1. What is Cypher?

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

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

Structure

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

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

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

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

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

Let’s see MATCH and RETURN in action.

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

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

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

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

Resulting in:

```
<table>
<thead>
<tr>
<th>john.name</th>
<th>fof.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;John&quot;</td>
<td>&quot;Maria&quot;</td>
</tr>
<tr>
<td>&quot;John&quot;</td>
<td>&quot;Steve&quot;</td>
</tr>
</tbody>
</table>
```

2 rows

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

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

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

Resulting in:

```
<table>
<thead>
<tr>
<th>user.name</th>
<th>follower.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Joe&quot;</td>
<td>&quot;Steve&quot;</td>
</tr>
<tr>
<td>&quot;John&quot;</td>
<td>&quot;Sara&quot;</td>
</tr>
</tbody>
</table>
```

2 rows

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

- **CREATE** (and **DELETE**): Create (and delete) nodes and relationships.
- **SET** (and **REMOVE**): Set values to properties and add labels on nodes using **SET** and use **REMOVE** to remove them.
1.2. Neo4j databases and graphs

This section describes databases and graphs in Neo4j.

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

- **DBMS**
  - A Neo4j server is a Database Management System capable of containing and managing multiple databases. Client applications will connect to a database server and be able to open sessions against a particular database.

- **Database**
  - A database is a storage and retrieval mechanism for collecting data in a defined space on disk and in memory. A client session provides access to a single database in the DBMS. Cypher commands executed through this session will apply to the default graph within that database. It is never possible to operate against multiple databases at the same time. Transactions are specific to a database.

- **Graph**
  - This is a data model within a database. In Neo4j 4.0 there is only one graph within each database, and many administrative commands that refer to a specific graph do so using the database name.
  - In Neo4j 4.0 Fabric it is possible to refer to multiple graphs within the same transaction and Cypher query.

Most of the time Cypher queries are reading or updating queries which are run against a graph. There are, however, administrative commands that apply to a database, or to the entire DBMS. Such 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 special system database.

More on this requirement is described in the chapter on Administration.

### 1.2.1. The System Database and the Default Database

All Neo4j servers will contain a built-in database called system which behaves differently than all other databases. In particular, when connected to this database you can only perform a specific set of administrative functions, as described in detail in the section on administration. Most of the available administrative commands can only be run by users with specific administrative privileges. Information on configuring the security privileges is described in the Operations Manual as well as the section below on security commands.

A fresh installation of Neo4j will include 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`.

### 1.2.2. Different editions of Neo4j

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

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

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

### 1.2.3. Limited Support Features

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

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

### 1.3. Querying, updating and administering

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

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

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

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

1.3.2. The structure of update queries

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

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

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

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

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

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

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

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

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

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

1.3.3. Returning data

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

After all the parts of the query comes one final `RETURN` clause. `RETURN` is not part of any query part — it is a period symbol at the end of a query. The `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.

### 1.4. Transactions

This section describes how Cypher queries work with database transactions.

All Cypher statements are explicitly run within a transaction. For read-only queries, the transaction will always succeed. For updating queries it is possible that a failure can occur for some reason, for example if the query attempts to violate a constraint, in which case the entire transaction is rolled back, and no changes are made to the graph. Every statement is executed within the context of the transaction, and nothing will be persisted to disk until that transaction is successfully committed.

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

While it is not possible to run a Cypher query outside a transaction, it is possible to run multiple queries within a single transaction using the following sequence of operations:

1. Open a transaction,
2. Run multiple updating Cypher queries.
3. Commit all of them in one go.

Note that the transaction will hold the changes in memory until the whole query, or whole set of queries, has finished executing. A query that makes a large number of updates will consequently use large amounts of memory. For memory configuration in Neo4j, see the [Neo4j Operations Manual](https://neo4j.com/docs/operations-manual/current/) → Memory configuration.

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

- For information on using transactions with a Neo4j driver, see the [Neo4j Driver manual](https://neo4j.com/docs/driver-manual/current/) → The session API.
- For information on using transactions over the HTTP API, see the [HTTP API documentation](https://neo4j.com/docs/http-api/current/) → Using the HTTP API.
- For information on using transactions within the embedded Core API, see the [Java Reference](https://neo4j.com/docs/java-reference/current/) → Executing Cypher queries from Java.

When writing procedures or using Neo4j embedded, remember that all iterators returned from an execution result should be either fully exhausted or closed. This ensures that the resources bound to them are properly released.
1.5. Cypher path matching

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

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

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

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

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

**Homomorphism**

No constraints for path matching.

**Node isomorphism**

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

**Relationship isomorphism**

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

1.5.1. Homomorphism

**Constraints:** No constraints for path matching.

**Example 1. Homomorphism**

The graph is composed of only two nodes \(a\) and \(b\), connected by one relationship, \((a:\text{Node})-\langle r:R\rangle\rightarrow(b:\text{Node})\).

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

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

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

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

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

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

**Example 2. Node isomorphism**

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

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

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

1.5.3. Relationship isomorphism

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

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

**Example 3. Relationship isomorphism**

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

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

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

1.5.4. Cypher path matching example

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

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

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

Query 1, create data.

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

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

Which gives us the following graph:

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

Query 2, friend of friends of Adam.

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

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

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

This is however not always desired. If the query should return the user, it is possible to spread the matching over multiple MATCH clauses, like so:
Query 3, multiple MATCH clauses.

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

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

Rows: 2

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

Query 4, equivalent to query 2.

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

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

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

Rows: 1
Chapter 2. Syntax

This section describes the syntax of the Cypher query language.

• Values and types
• Naming rules and recommendations
• Expressions
  ◦ Expressions in general
  ◦ Note on string literals
  ◦ CASE Expressions
• Variables
• Reserved keywords
• Parameters
  ◦ String literal
  ◦ Regular expression
  ◦ Case-sensitive string pattern matching
  ◦ Create node with properties
  ◦ Create multiple nodes with properties
  ◦ Setting all properties on a node
  ◦ SKIP and LIMIT
  ◦ Node id
  ◦ Multiple node ids
  ◦ Calling procedures
• Operators
  ◦ Operators at a glance
  ◦ Aggregation operators
  ◦ Property operators
  ◦ Mathematical operators
  ◦ Comparison operators
  ◦ Boolean operators
  ◦ String operators
  ◦ Temporal operators
  ◦ Map operators
  ◦ List operators
• Comments
• Patterns
  ◦ Patterns for nodes
  ◦ Patterns for related nodes
  ◦ Patterns for labels
  ◦ Specifying properties
  ◦ Patterns for relationships
  ◦ Variable-length pattern matching
  ◦ Assigning to path variables
• Temporal (Date/Time) values
  ◦ Introduction
  ◦ Time zones
  ◦ Temporal instants
    ▪ Specifying temporal instants
      ▪ Specifying dates
      ▪ Specifying times
      ▪ Specifying time zones
      ▪ Examples
      ▪ Accessing components of temporal instants
  ◦ Durations
    ▪ Specifying durations
      ▪ Examples
      ▪ Accessing components of durations
  ◦ Examples
  ◦ Temporal indexing
• Spatial values
  ◦ Introduction
  ◦ Coordinate Reference Systems
    ▪ Geographic coordinate reference systems
    ▪ Cartesian coordinate reference systems
  ◦ Spatial instants
    ▪ Creating points
    ▪ Accessing components of points
  ◦ Spatial index
• Lists
  ◦ Lists in general
2.1. Values and types

This section provides an overview of data types in Cypher.

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

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

Property types
- Integer, Float, String, Boolean, Point, Date, Time, LocalDateTime, and Duration.

Structural types
- Node, Relationship, and Path.

Composite types
- List and Map.

2.1.1. Property types

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

The property types:
- **Number**, an abstract type, which has the subtypes **Integer** and **Float**
- **String**
- **Boolean**
• The spatial type Point

• Temporal types: Date, Time, LocalTime, DateTime, LocalDateTime and Duration

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

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

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

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

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

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

### 2.1.2. Structural types

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

The structural types:

- **Node**
  - **Id**
  - **Label(s)**

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

  - **Map (of properties)**

- **Relationship**
2.1.3. Composite types

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

The composite types:

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

- **Map**, a heterogeneous, unordered collection of (Key, Value) pairs.
  - Key is a String
  - Value has any property, structural or composite type

Composite values can also contain **null**.

Special care must be taken when using **null** (see Working with **null**).

2.2. Naming rules and recommendations

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

2.2.1. Naming rules

- **Alphabetic characters**:
  - Names should begin with an alphabetic character.
This includes "non-English" characters, such as å, ä, ö, ü etc.

- **Numbers:**
  - Names should not begin with a number.
  - To illustrate, 1first is not allowed, whereas first1 is allowed.

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

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

- **Case-sensitive:**
  - Names are case-sensitive and thus, :PERSON, :Person and :person are three different labels, and n and N are two different variables.

- **Whitespace characters:**
  - Leading and trailing whitespace characters will be removed automatically. For example, MATCH ( a ) RETURN a is equivalent to MATCH (a) RETURN a.

---

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

---

### 2.2.2. Scoping and namespace rules

- **Node labels, relationship types and property names may re-use names.**
  - The following query — with a for the label, type and property name — is valid: CREATE (a:a {a: 'a'})-[r:a]->(b:a {a: 'a'}).

- **Variables for nodes and relationships must not re-use names within the same query scope.**
  - The following query is not valid as the node and relationship both have the name a: CREATE (a)-[a]->(b).

### 2.2.3. Recommendations

Here are the recommended naming conventions:

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

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

- Expressions in general
- Note on string literals
- **CASE expressions**
  - Simple CASE form: comparing an expression against multiple values
  - Generic CASE form: allowing for multiple conditionals to be expressed
  - Distinguishing between when to use the simple and generic CASE forms

2.3.1. Expressions in general

<table>
<thead>
<tr>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
</tr>
</tbody>
</table>

An expression in Cypher can be:

- A decimal (integer or float) literal: 13, -40000, 3.14, 6.022E23.
- A hexadecimal integer literal (starting with 0x): 0x13af, 0xFC3A9, -0x66eff.
- An octal integer literal (starting with 0): 01372, 02127, -05671.
- A string literal: 'Hello', "World".
- A boolean literal: true, false.
- A variable: n, x, rel, myFancyVariable, 'A name with weird stuff in it[]!'.
- A property: n.prop, x.prop, rel.thisProperty, myFancyVariable."(weird property name)".
- A dynamic property: n["prop"], rel[n.city + n.zip], map[coll[0]].
- A parameter: $param, $0.
- A list of expressions: ['a', 'b'], [1, 2, 3], ['a', 2, n.property, $param], [].
- A function call: length(p), nodes(p).
- An aggregate function: avg(x.prop), count(*)
- A path-pattern: (a)-[r]->(b), (a)-[r]-(b), (a)--(b), (a)-->(b)<--(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, exists(a.name).
- An existential subquery is an expression that returns true or false: EXISTS { MATCH (n)-[r]->(p) WHERE p.name = 'Sven' }.
- A regular expression: a.name =~ 'Tim.*'.
- A case-sensitive string matching expression: a.surname STARTS WITH 'Sven', a.surname ENDS WITH
\[ \text{son'} \text{ or a.surname CONTAINS 'son'}. \]

- A **CASE** expression.

### 2.3.2. Note on string literals

String literals can contain the following escape sequences:

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

### 2.3.3. **CASE** expressions

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

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

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

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

**Query**

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

**Table 1. Result**

<table>
<thead>
<tr>
<th>Array</th>
<th>Name</th>
<th>Eyes</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>['one', 'two', 'three']</td>
<td>Eskil</td>
<td>blue</td>
<td>41</td>
</tr>
</tbody>
</table>
Generic **CASE** form: allowing for multiple conditionals to be expressed

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

**Syntax:**

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

**Arguments:**

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

**Query**

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

**Table 2. Result**

<table>
<thead>
<tr>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
Distinguishing between when to use the simple and generic `CASE` forms

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

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

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

Table 3. Result

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

Rows: 5

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

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

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

Table 4. Result

<table>
<thead>
<tr>
<th>n.name</th>
<th>age_10_years_ago</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Alice&quot;</td>
<td>28</td>
</tr>
<tr>
<td>&quot;Bob&quot;</td>
<td>15</td>
</tr>
<tr>
<td>&quot;Eskil&quot;</td>
<td>31</td>
</tr>
</tbody>
</table>
Using the result of **CASE** in the succeeding clause or statement

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

**Query**

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

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

**Table 5. Result**

<table>
<thead>
<tr>
<th>(empty result)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 0</td>
</tr>
<tr>
<td>Properties set: 5</td>
</tr>
</tbody>
</table>

### 2.4. Variables

This section provides an overview of variables in Cypher.

When you reference parts of a pattern or a query, you do so by naming them. The names you give the different parts are called variables.

In this example:

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

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

Information regarding the naming of variables may be found [here](#).
Variables are only visible in the same query part
Variables are not carried over to subsequent queries. If multiple query parts are chained together using `WITH`, variables have to be listed in the `WITH` clause to be carried over to the next part. For more information see `WITH`.

2.5. Reserved keywords

This section contains a list of reserved keywords in Cypher.

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

The reserved keywords are not permitted to be used as identifiers in the following contexts:

- Variables
- Function names
- Parameters

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

2.5.1. Clauses

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

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

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

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

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

2.5.6. Schema
• CONSTRAINT
• CREATE
• DROP
• EXISTS
• INDEX
• NODE
• KEY
• UNIQUE

2.5.7. Hints
• INDEX
• JOIN
• PERIODIC
• COMMIT
• SCAN
• USING

2.5.8. Literals
• false
• null
• true

2.5.9. Reserved for future use
• ADD
• DO
• FOR
• MANDATORY
• OF
• REQUIRE
2.6. Parameters

This section describes parameterized querying.

2.6.1. Introduction

Cypher supports querying with parameters. A parameterized query is a query in which placeholders are used for parameters and the parameter values are supplied at execution time. This means developers do not have to resort to string building to create a query. Additionally, parameters make caching of execution plans much easier for Cypher, thus leading to faster query execution times.

Parameters can be used for:

- literals and expressions
- node and relationship ids

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

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

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

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

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

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

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

Parameters

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

Query

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

You can use parameters in this syntax as well:

Parameters

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

Query

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

2.6.3. Regular expression

Parameters

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

Query

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

2.6.4. Case-sensitive string pattern matching

Parameters

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

Query

```cypher
MATCH (n:Person)
WHERE n.name STARTS WITH $name
RETURN n.name
```
2.6.5. Create node with properties

Parameters

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

Query

`CREATE ($props)`

2.6.6. Create multiple nodes with properties

Parameters

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

Query

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

2.6.7. Setting all properties on a node

Note that this will replace all the current properties.

Parameters

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

Query

```sql
MATCH (n:Person)
WHERE n.name = 'Michaela'
SET n = $props
```
2.6.8. **SKIP and LIMIT**

**Parameters**

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

**Query**

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

2.6.9. **Node id**

**Parameters**

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

**Query**

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

2.6.10. **Multiple node ids**

**Parameters**

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

**Query**

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

2.6.11. **Calling procedures**

**Parameters**

```
{
  "indexname" : "My index"
}
```
CALL db.resampleIndex($indexname)

2.7. Operators

- Operators at a glance

- Aggregation operators
  - Using the **DISTINCT** operator

- Property operators
  - Statically accessing a property of a node or relationship using the **.`** operator
  - Filtering on a dynamically-computed property key using the **`\ operator**
  - Replacing all properties of a node or relationship using the **`=` operator**
  - Mutating specific properties of a node or relationship using the **`+=` operator**

- Mathematical operators
  - Using the exponentiation operator **`^`**
  - Using the unary minus operator **`-`**

- Comparison operators
  - Comparing two numbers
  - Using **`STARTS WITH`** to filter names
  - Equality and comparison of values
  - Ordering and comparison of values
  - Chaining comparison operations
  - Using a regular expression with **`=~`** to filter words

- Boolean operators
  - Using boolean operators to filter numbers

- String operators
  - Concatenating two strings using **`+`**

- Temporal operators
  - Adding and subtracting a **`Duration`** to or from a temporal instant
  - Adding and subtracting a **`Duration`** to or from another **`Duration`**
  - Multiplying and dividing a **`Duration`** with or by a number

- Map operators
  - Statically accessing the value of a nested map by key using the **`. operator"**
  - Dynamically accessing the value of a map by key using the **`\ operator and a parameter`**

- List operators
• Concatenating two lists using +
• Using IN to check if a number is in a list
• Using IN for more complex list membership operations
• Accessing elements in a list using the [\ operator]
• Dynamically accessing an element in a list using the [\ operator and a parameter]
• Using IN with [\ on a nested list]

2.7.1. Operators at a glance

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

2.7.2. Aggregation operators

The aggregation operators comprise:

• remove duplicates values: DISTINCT

Using the DISTINCT operator

Retrieve the unique eye colors from Person nodes.

Query

```sql
CREATE
(a:Person {name: 'Anne', eyeColor: 'blue'}),
(b:Person {name: 'Bill', eyeColor: 'brown'}),
(c:Person {name: 'Carol', eyeColor: 'blue'})
WITH [a, b, c] AS ps
UNWIND ps AS p
RETURN DISTINCT p.eyeColor
```
Even though both 'Anne' and 'Carol' have blue eyes, 'blue' is only returned once.

Table 6. Result

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

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

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

2.7.3. Property operators

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

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

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

Query

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

Table 7. Result

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

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

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

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

Table 8. Result

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

Rows: 1
Nodes created: 4
Properties set: 8
Labels added: 4

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

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

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

Query

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

All the existing properties on the node are replaced by those provided in the map; i.e. the `name` property is updated from `Jane` to `Ellen`, the `age` property is deleted, and the `livesIn` property is added.

Table 9. Result

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

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

See [Replace all properties using a map and =](#) for more details on using the property replacement operator `=`.
Mutating specific properties of a node or relationship using the `+=` operator

**Query**

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

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

**Table 10. Result**

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

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

See [Mutate specific properties using a map and `+=` for more details on using the property mutation operator `+=`].

### 2.7.4. Mathematical operators

The mathematical operators comprise:

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

**Using the exponentiation operator `^`**

**Query**

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

**Table 11. Result**

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

Rows: 1
Using the unary minus operator -

Query

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

Table 12. Result

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

Rows: 1

2.7.5. Comparison operators

The comparison operators comprise:

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

String-specific comparison operators comprise:

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

Comparing two numbers

Query

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

Table 13. Result

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

Rows: 1
See Equality and comparison of values for more details on the behavior of comparison operators, and Using ranges for more examples showing how these may be used.

Using STARTS WITH to filter names

Query

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

Table 14. Result

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

Rows: 2

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

Equality and comparison of values

Equality

Cypher supports comparing values (see Values and types) by equality using the = and <> operators.

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

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

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

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

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

It is an error to compare values that cannot be compared.
Ordering and comparison of values

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

- Numerical values are compared for ordering using numerical order (e.g. $3 < 4$ is true).
- The special value `java.lang.Double.NaN` is regarded as being larger than all other numbers.
- String values are compared for ordering using lexicographic order (e.g. "x" < "xy").
- Boolean values are compared for ordering such that `false < true`.

Comparison of spatial values:

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

Ordering of spatial values:

- `ORDER BY` requires all values to be orderable.
- Points are ordered after arrays and before temporal types.
- Points of different CRS are ordered by the CRS code (the value of SRID field). For the currently supported set of Coordinate Reference Systems this means the order: 4326, 4979, 7302, 9157.
- Points of the same CRS are ordered by each coordinate value in turn, $x$ first, then $y$ and finally $z$.
- Note that this order is different to the order returned by the spatial index, which will be the order of the space filling curve.

Comparison of temporal values:

- **Temporal instant values** are comparable within the same type. An instant is considered less than another instant if it occurs before that instant in time, and it is considered greater than if it occurs after.
- Instant values that occur at the same point in time — but that have a different time zone — are not considered equal, and must therefore be ordered in some predictable way. Cypher prescribes that, after the primary order of point in time, instant values be ordered by effective time zone offset, from west (negative offset from UTC) to east (positive offset from UTC). This has the effect that times that represent the same point in time will be ordered with the time with the earliest local time first. If two instant values represent the same point in time, and have the same time zone offset, but a different named time zone (this is possible for `DateTime` only, since `Time` only has an offset), these values are not considered equal, and ordered by the time zone identifier, alphabetically, as its third ordering component.
- **Duration** values cannot be compared, since the length of a day, month or year is not known without knowing which day, month or year it is. Since Duration values are not comparable, the result of applying a comparison operator between two Duration values is `null`. If the type, point in
time, offset, and time zone name are all equal, then the values are equal, and any difference in order is impossible to observe.

- **Ordering of temporal values:**
  - `ORDER BY` requires all values to be orderable.
  - Temporal instances are ordered after spatial instances and before strings.
  - Comparable values should be ordered in the same order as implied by their comparison order.
  - Temporal instant values are first ordered by type, and then by comparison order within the type.
  - Since no complete comparison order can be defined for `Duration` values, we define an order for `ORDER BY` specifically for `Duration`:
    - `Duration` values are ordered by normalising all components as if all years were 365.2425 days long (`PT8765H49M12S`), all months were 30.436875 (1/12 year) days long (`PT730H29M06S`), and all days were 24 hours long \[1\].

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

### 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:

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

is equivalent to

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

Thus it will match all nodes where the age is between 21 and 30.

This syntax extends to all equality and inequality comparisons, as well as extending to chains longer than three.

For example:

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

Is equivalent to:

```sql
a < b AND b = c AND c <= d AND d <> e
```
The 365.2425 days per year comes from the frequency of leap years. A leap year occurs on a year with an ordinal number divisible by 4, that is not divisible by 100, unless it divisible by 400. This means that over 400 years there are \( ((365 \times 4 + 1) \times 25 - 1) \times 4 + 1 = 146097 \) days, which means an average of 365.2425 days per year.
Chapter 3. Using a regular expression with =~ to filter words

Query

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

Table 15. Result

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

Rows: 2

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

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

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

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

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

4.1. Using boolean operators to filter numbers

**Query**

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

**Table 16. Result**

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

Rows: 3
Chapter 5. String operators

The string operators comprise:

- concatenating strings: +

5.1. Concatenating two strings with +

Query

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

Table 17. Result

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

Rows: 1
Chapter 6. Temporal operators

Temporal operators comprise:

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

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

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

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

Query

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

Table 18. Result

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

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

Query

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

Table 19. Result

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

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

Query

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

Table 20. Result

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

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

Query

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

Table 21. Result

<table>
<thead>
<tr>
<th>duration1</th>
<th>duration2</th>
<th>duration1 + duration2</th>
<th>duration1 - duration2</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1Y2M5DT16H13M10.000000001S</td>
<td>P1M-14DT15H49M10S</td>
<td>P1Y2YMT32H2M20.00000001S</td>
<td>P1Y4M28DT24M0.00000001S</td>
</tr>
</tbody>
</table>

Rows: 1
6.3. Multiplying and dividing a Duration with or by a number

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

Query

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

Table 22. Result

<table>
<thead>
<tr>
<th>aDuration</th>
<th>aDuration * 2</th>
<th>aDuration / 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P14DT13M10.000000001S</td>
<td>P28DT26M20.000000002S</td>
<td>P4DT16H4M23.333333333S</td>
</tr>
</tbody>
</table>

Rows: 1
Chapter 7. Map operators

The map operators comprise:

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

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

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

Query

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

Table 23. Result

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

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

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

Parameters

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

Query

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

Table 24. Result

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

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

The list operators comprise:

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

The behavior of the \texttt{IN} and \texttt{[]} operators with respect to \texttt{null} is detailed here.

8.1. Concatenating two lists using $+$

Query

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

Table 25. Result

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

8.2. Using \texttt{IN} to check if a number is in a list

Query

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

Table 26. Result

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

8.3. Using \texttt{IN} for more complex list membership operations

The general rule is that the \texttt{IN} operator will evaluate to \texttt{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 $\text{innerList}$ are compared pairwise in ascending order from the first element in $\text{innerList}$ to the last element in $\text{innerList}$.
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 \texttt{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 \texttt{false}.

**Table 27. Result**

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

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

**Query**

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

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

**Table 28. Result**

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

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

### 8.4. 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 29. Result**

<table>
<thead>
<tr>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>[&quot;John&quot;,&quot;Bill&quot;]</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

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

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

**Parameters**

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

**Query**

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

**Table 30. Result**

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

**8.6. Using IN with [ ] on a nested list**

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

**Parameters**

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

**Query**

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

**Table 31. Result**

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

8.7. Comments

This section describes how to use comments in Cypher.

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

Examples:

```
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
```

8.8. Patterns

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

8.8.1. Introduction

Patterns and pattern-matching are at the very heart of Cypher, so being effective with Cypher requires a good understanding of patterns.

Using patterns, you describe the shape of the data you’re looking for. For example, in the MATCH clause you describe the shape with a pattern, and Cypher will figure out how to get that data for you.

The pattern describes the data using a form that is very similar to how one typically draws the shape of property graph data on a whiteboard: usually as circles (representing nodes) and arrows between them to
represent relationships.

Patterns appear in multiple places in Cypher: in MATCH, CREATE and MERGE clauses, and in pattern expressions. Each of these is described in more detail in:

- MATCH
- OPTIONAL MATCH
- CREATE
- MERGE
- Using path patterns in WHERE

8.8.2. Patterns for nodes

The very simplest 'shape' that can be described in a pattern is a node. A node is described using a pair of parentheses, and is typically given a name. For example:

(a)

This simple pattern describes a single node, and names that node using the variable \( a \).

8.8.3. Patterns for related nodes

A more powerful construct is a pattern that describes multiple nodes and relationships between them. Cypher patterns describe relationships by employing an arrow between two nodes. For example:

(a)-->b

This pattern describes a very simple data shape: two nodes, and a single relationship from one to the other. In this example, the two nodes are both named as \( a \) and \( b \) respectively, and the relationship is 'directed': it goes from \( a \) to \( b \).

This manner of describing nodes and relationships can be extended to cover an arbitrary number of nodes and the relationships between them, for example:

(a)-->b<--(c)

Such a series of connected nodes and relationships is called a "path".

Note that the naming of the nodes in these patterns is only necessary should one need to refer to the same node again, either later in the pattern or elsewhere in the Cypher query. If this is not necessary, then the name may be omitted, as follows:

(a)-->()<--(c)
8.8.4. Patterns for labels

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

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

One can also describe a node that has multiple labels:

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

8.8.5. Specifying properties

Nodes and relationships are the fundamental structures in a graph. Neo4j uses properties on both of these to allow for far richer models.

Properties can be expressed in patterns using a map-construct: curly brackets surrounding a number of key-expression pairs, separated by commas. E.g. a node with two properties on it would look like:

```
(a {name: 'Andy', sport: 'Brazilian Ju-Jitsu'})
```

A relationship with expectations on it is given by:

```
(a)-[[blocked: false]]->(b)
```

When properties appear in patterns, they add an additional constraint to the shape of the data. In the case of a CREATE clause, the properties will be set in the newly-created nodes and relationships. In the case of a MERGE clause, the properties will be used as additional constraints on the shape any existing data must have (the specified properties must exactly match any existing data in the graph). If no matching data is found, then MERGE behaves like CREATE and the properties will be set in the newly created nodes and relationships.

Note that patterns supplied to CREATE may use a single parameter to specify properties, e.g: CREATE (node $paramName). This is not possible with patterns used in other clauses, as Cypher needs to know the property names at the time the query is compiled, so that matching can be done effectively.

8.8.6. Patterns for relationships

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

```
(a)--(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:

```plaintext
(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:

```plaintext
(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:

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

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

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

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

### 8.8.7. Variable-length pattern matching

Variable length pattern matching in versions 2.1.x and earlier does not enforce relationship uniqueness for patterns described within a single MATCH clause. This means that a query such as the following: MATCH (a)-[r]->(b), p = (a)-[]->(c) RETURN *, relationships(p) AS rs may include r as part of the rs set. This behavior has changed in versions 2.2.0 and later, in such a way that r will be excluded from the result set, as this better adheres to the rules of relationship uniqueness as documented here Cypher path matching. If you have a query pattern that needs to retrace relationships rather than ignoring them as the relationship uniqueness rules normally dictate, you can accomplish this using multiple match clauses, as follows: MATCH (a)-[r]->(b) MATCH p = (a)-[]->(c) RETURN *, relationships(p). This will work in all versions of Neo4j that support the MATCH clause, namely 2.0.0 and later.

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

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

This describes a graph of three nodes and two relationship, all in one path (a path of length 2). This is equivalent to:
A range of lengths can also be specified: such relationship patterns are called 'variable length relationships'. For example:

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

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

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

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

To describe paths of length 5 or less, use:

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

Both bounds can be omitted, allowing paths of any length to be described:

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

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

Graph

Query

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

Table 32. Result

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

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

8.8.8. Assigning to path variables

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

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

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

8.9. Temporal (Date/Time) values

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

- Introduction
- Time zones
- Temporal instants
  - Specifying temporal instants
  - Specifying dates
  - Specifying times
  - Specifying time zones
  - Examples
  - Accessing components of temporal instants
- Durations
  - Specifying durations
  - Examples
  - Accessing components of durations
- Examples
- Temporal indexing
Refer to Temporal functions - 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.

8.9.1. Introduction

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

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

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

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

8.9.2. Time zones

Time zones are represented either as an offset from UTC, or as a logical identifier of a named time zone (these are based on the IANA time zone database). 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)

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

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

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

8.9.3. Temporal instants

Specifying temporal instants

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

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

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

Specifying dates

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

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

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

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

The following formats are supported for specifying dates:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
<th>Interpretation of example</th>
</tr>
</thead>
<tbody>
<tr>
<td>YYYY-MM-DD</td>
<td>Calendar date: Year-Month-Day</td>
<td>2015-07-21</td>
<td>2015-07-21</td>
</tr>
<tr>
<td>YYYYMMDD</td>
<td>Calendar date: Year-Month-Day</td>
<td>20150721</td>
<td>2015-07-21</td>
</tr>
<tr>
<td>YYYY-MM</td>
<td>Calendar date: Year-Month</td>
<td>2015-07</td>
<td>2015-07-01</td>
</tr>
<tr>
<td>YYYYMM</td>
<td>Calendar date: Year-Month</td>
<td>201507</td>
<td>2015-07-01</td>
</tr>
<tr>
<td>YYYY-Ww-D</td>
<td>Week date: Year-Week-Day</td>
<td>2015-W30-2</td>
<td>2015-07-21</td>
</tr>
<tr>
<td>Format</td>
<td>Description</td>
<td>Example</td>
<td>Interpretation of example</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------</td>
<td>------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>YYYY-WwwD</td>
<td>Week date: Year-Week-Day</td>
<td>2015#302</td>
<td>2015-07-21</td>
</tr>
<tr>
<td>YYYY-Www</td>
<td>Week date: Year-Week</td>
<td>2015-W30</td>
<td>2015-07-20</td>
</tr>
<tr>
<td>YYYY-Www</td>
<td>Week date: Year-Week</td>
<td>2015W30</td>
<td>2015-07-20</td>
</tr>
<tr>
<td>YYYY-Qq-DD</td>
<td>Quarter date: Year-Quarter-Day</td>
<td>2015-Q2-60</td>
<td>2015-05-30</td>
</tr>
<tr>
<td>YYYYQqDD</td>
<td>Quarter date: Year-Quarter</td>
<td>2015Q260</td>
<td>2015-05-30</td>
</tr>
<tr>
<td>YYYY-Qq</td>
<td>Quarter date: Year-Quarter</td>
<td>2015-Q2</td>
<td>2015-04-01</td>
</tr>
<tr>
<td>YYYYQq</td>
<td>Quarter date: Year-Quarter</td>
<td>2015Q2</td>
<td>2015-04-01</td>
</tr>
<tr>
<td>YYYY-DDD</td>
<td>Ordinal date: Year-Day</td>
<td>2015-202</td>
<td>2015-07-21</td>
</tr>
<tr>
<td>YYYYDDDD</td>
<td>Ordinal date: Year-Day</td>
<td>2015202</td>
<td>2015-07-21</td>
</tr>
<tr>
<td>YYYY</td>
<td>Year</td>
<td>2015</td>
<td>2015-01-01</td>
</tr>
</tbody>
</table>

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

Specifying times

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

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

The following formats are supported for specifying times:
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 (ː).
- The time zone of the International Date Line is denoted either by +12:00 or −12:00, depending on country.

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

The following formats are supported for specifying time zones:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
<th>Supported for DateTime</th>
<th>Supported for Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>UTC</td>
<td>Z</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Examples

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

Parsing a `DateTime` using the calendar date format:

Query

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

Table 33. Result

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

Rows: 1

Parsing a `LocalDateTime` using the ordinal date format:

Query

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

Table 34. Result

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

Rows: 1

Parsing a `Date` using the week date format:
Query

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

Table 35. Result

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

Rows: 1

Parsing a Time:

Query

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

Table 36. Result

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

Rows: 1

Parsing a LocalTime:

Query

```
RETURN localtime('12:50:35.56') AS theLocalTime
```

Table 37. Result

<table>
<thead>
<tr>
<th>theLocalTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:50:35.56</td>
</tr>
</tbody>
</table>

Rows: 1

Accessing components of temporal instants

Components of temporal instant values can be accessed as properties.

Table 38. Components of temporal instant values and where they are supported

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
<th>Type</th>
<th>Range/Format</th>
<th>Date</th>
<th>DateTime</th>
<th>LocalDateTime</th>
<th>Time</th>
<th>LocalTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>instant.year</td>
<td>The year component represents the astronomical year number of the instant.</td>
<td>Integer</td>
<td>At least 4 digits. For more information, see the rules for using the Year component</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Component</td>
<td>Description</td>
<td>Type</td>
<td>Range/Format</td>
<td>Date</td>
<td>Date Time</td>
<td>Local Date Time</td>
<td>Time</td>
<td>Local Time</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------</td>
<td>--------------</td>
<td>------</td>
<td>-----------</td>
<td>-----------------</td>
<td>------</td>
<td>------------</td>
</tr>
<tr>
<td>instant.quarter</td>
<td>The quarter-of-the-year component.</td>
<td>Integer</td>
<td>1 to 4</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>instant.month</td>
<td>The month-of-the-year component.</td>
<td>Integer</td>
<td>1 to 12</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>instant.week</td>
<td>The week-of-the-year component.</td>
<td>Integer</td>
<td>1 to 53</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>instant.weekYear</td>
<td>The year that the week-of-year component belongs to.(^{(i)})</td>
<td>Integer</td>
<td>At least 4 digits. For more information, see the rules for using the Year component</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>instant.dayOfQuarter</td>
<td>The day-of-the-quarter component.</td>
<td>Integer</td>
<td>1 to 92</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>instant.quarterDay</td>
<td>The day-of-the-quarter component. (alias for instant.dayOfQuarter)</td>
<td>Integer</td>
<td>1 to 92</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>instant.day</td>
<td>The day-of-the-month component.</td>
<td>Integer</td>
<td>1 to 31</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>instant.ordinalDay</td>
<td>The day-of-the-year component.</td>
<td>Integer</td>
<td>1 to 366</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>instant.dayOfWeek</td>
<td>The day-of-the-week component (the first day of the week is Monday).</td>
<td>Integer</td>
<td>1 to 7</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>instant.weekDay</td>
<td>The day-of-the-week component (alias for instant.dayOfWeek)</td>
<td>Integer</td>
<td>1 to 7</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>instant.hour</td>
<td>The hour component.</td>
<td>Integer</td>
<td>0 to 23</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>instant.minute</td>
<td>The minute component.</td>
<td>Integer</td>
<td>0 to 59</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>instant.second</td>
<td>The second component.</td>
<td>Integer</td>
<td>0 to 59</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>instant.Millisecond</td>
<td>The millisecond component.</td>
<td>Integer</td>
<td>0 to 999</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The following query shows how to extract the components of a Date value:

Query

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

Table 39. Result

<table>
<thead>
<tr>
<th>d.year</th>
<th>d.quarter</th>
<th>d.month</th>
<th>d.week</th>
<th>d.weekYear</th>
<th>d.day</th>
<th>d.ordinalDay</th>
<th>d.dayOfWeek</th>
<th>d.dayOfQuarter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984</td>
<td>4</td>
<td>10</td>
<td>41</td>
<td>1984</td>
<td>11</td>
<td>285</td>
<td>4</td>
<td>11</td>
</tr>
</tbody>
</table>

Rows: 1

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

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

### Table 41. Result

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

Rows: 1

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

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

### Table 42. Result

<table>
<thead>
<tr>
<th>d.timezone</th>
<th>d.offset</th>
<th>d.offsetMinutes</th>
<th>d.epochSeconds</th>
<th>d.epochMillis</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Europe/Stockholm&quot;</td>
<td>&quot;+01:00&quot;</td>
<td>60</td>
<td>469020674</td>
<td>469020674645</td>
</tr>
</tbody>
</table>

Rows: 1

8.9.4. Durations
Specifying durations

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

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

- **Unit-based form:** \texttt{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 \texttt{n} denotes a numeric value which can be arbitrarily large.
  - The value of the last — and least significant — component may contain a decimal fraction.
  - Each component must be suffixed by a component identifier denoting the unit.
  - The unit-based form uses \texttt{M} as a suffix for both months and minutes. Therefore, time parts must always be preceded with \texttt{T}, even when no components of the date part are given.

- **Date-and-time-based form:** \texttt{P<date>T<time>}
  - Unlike the unit-based form, this form requires each component to be within the bounds of a valid LocalDateTime.

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

<table>
<thead>
<tr>
<th>Component identifier</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>Years</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>Months</td>
<td>Must be specified before \texttt{T}</td>
</tr>
<tr>
<td>W</td>
<td>Weeks</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>Days</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>Hours</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>Minutes</td>
<td>Must be specified after \texttt{T}</td>
</tr>
<tr>
<td>S</td>
<td>Seconds</td>
<td></td>
</tr>
</tbody>
</table>

Examples

The following examples demonstrate various methods of parsing Duration values. For more details, refer to \textit{Creating a Duration from a string}.

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

Query

\begin{verbatim}
RETURN duration\( \text{"P14DT16H12M"} \) AS theDuration
\end{verbatim}
Table 43. Result

<table>
<thead>
<tr>
<th>theDuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>P14DT16H12M</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Rows: 1

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

Query

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

Table 44. Result

<table>
<thead>
<tr>
<th>theDuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>P5M1DT12H</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Rows: 1

Return a Duration of 45 seconds:

Query

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

Table 45. Result

<table>
<thead>
<tr>
<th>theDuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>PT45S</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Rows: 1

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

Query

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

Table 46. Result

<table>
<thead>
<tr>
<th>theDuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>P17DT12H</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

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

Please note that:

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

It is also possible to access the smaller (less significant) components of a component group bounded by the largest (most significant) component of the group:
<table>
<thead>
<tr>
<th>Component</th>
<th>Component Group</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>duration.quartersOfYear</td>
<td>Months</td>
<td>The number of quarters in the group that do not make a whole year</td>
<td>Integer</td>
</tr>
<tr>
<td>duration.monthsOfYear</td>
<td>Months</td>
<td>The number of months in the group that do not make a whole year</td>
<td>Integer</td>
</tr>
<tr>
<td>duration.monthsOfQuarter</td>
<td>Months</td>
<td>The number of months in the group that do not make a whole quarter</td>
<td>Integer</td>
</tr>
<tr>
<td>duration.daysOfWeek</td>
<td>Days</td>
<td>The number of days in the group that do not make a whole week</td>
<td>Integer</td>
</tr>
<tr>
<td>duration.minutesOfHour</td>
<td>Seconds</td>
<td>The number of minutes in the group that do not make a whole hour</td>
<td>Integer</td>
</tr>
<tr>
<td>duration.secondsOfMinute</td>
<td>Seconds</td>
<td>The number of seconds in the group that do not make a whole minute</td>
<td>Integer</td>
</tr>
<tr>
<td>duration.millisecondsOfSecond</td>
<td>Seconds</td>
<td>The number of milliseconds in the group that do not make a whole second</td>
<td>Integer</td>
</tr>
<tr>
<td>duration.microsecondsOfSecond</td>
<td>Seconds</td>
<td>The number of microseconds in the group that do not make a whole second</td>
<td>Integer</td>
</tr>
<tr>
<td>duration.nanosecondsOfSecond</td>
<td>Seconds</td>
<td>The number of nanoseconds in the group that do not make a whole second</td>
<td>Integer</td>
</tr>
</tbody>
</table>

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

**Query**

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

**Table 47. Result**

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

Rows: 1

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

**Query**

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

**Table 48. Result**

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

Rows: 1
The following query shows how to extract the most significant second based components of a `Duration` value:

Query

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

Table 49. Result

<table>
<thead>
<tr>
<th>d.hours</th>
<th>d.minutes</th>
<th>d.seconds</th>
<th>d.milliseconds</th>
<th>d.microseconds</th>
<th>d.nanoseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>61</td>
<td>3661</td>
<td>36611111</td>
<td>3661111111</td>
<td>366111111111</td>
</tr>
</tbody>
</table>

Rows: 1

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 50. Result

<table>
<thead>
<tr>
<th>d.minutesOfHour</th>
<th>d.secondsOfMinute</th>
<th>d.millisecondsOfSecond</th>
<th>d.microsecondsOfSecond</th>
<th>d.nanosecondsOfSecond</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>111</td>
<td>11111</td>
<td>111111111</td>
</tr>
</tbody>
</table>

Rows: 1

8.9.5. Examples

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

Create a `Duration` representing 1.5 days:

Query

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

Table 51. Result

<table>
<thead>
<tr>
<th>theDuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1DT12H</td>
</tr>
</tbody>
</table>

Rows: 1
Compute the Duration between two temporal instants:

Query

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

Table 52. Result

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

Compute the number of days between two Date values:

Query

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

Table 53. Result

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

Get the first Date of the current year:

Query

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

Table 54. Result

<table>
<thead>
<tr>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>2022-01-01</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

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

Query

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

Table 55. Result

<table>
<thead>
<tr>
<th>thursday</th>
</tr>
</thead>
<tbody>
<tr>
<td>2019-10-03</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

Get the Date of the last day of the next month:
### Add a Duration to a Date:

**Query**

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

**Table 56. Result**

<table>
<thead>
<tr>
<th>lastDay</th>
<th>2022-09-30</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows:</td>
<td>1</td>
</tr>
</tbody>
</table>

### Add a Duration to a Date:

**Query**

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

**Table 57. Result**

<table>
<thead>
<tr>
<th>theTime</th>
<th>01:42:19Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows:</td>
<td>1</td>
</tr>
</tbody>
</table>

### Add two Duration values:

**Query**

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

**Table 58. Result**

<table>
<thead>
<tr>
<th>theDuration</th>
<th>P1M2DT25H</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows:</td>
<td>1</td>
</tr>
</tbody>
</table>

### Multiply a Duration by a number:

**Query**

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

**Table 59. Result**

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

### Divide a Duration by a number:
Query

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

Table 60. Result

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

Examine whether two instants are less than one day apart:

Query

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

Table 61. Result

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

Return the abbreviated name of the current month:

Query

```
```

Table 62. Result

<table>
<thead>
<tr>
<th>month</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Aug&quot;</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

8.9.6. Temporal indexing

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

8.10. Spatial values

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

- Introduction
- Coordinate Reference Systems
  - Geographic coordinate reference systems
  - Cartesian coordinate reference systems
- Spatial instants
  - Creating points
  - Accessing components of points
- Spatial index
- Comparability and Orderability

Refer to Spatial functions for information regarding spatial functions allowing for the creation and manipulation of spatial values.

Refer to Ordering and comparison of values for information regarding the comparison and ordering of spatial values.

8.10.1. Introduction

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

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

8.10.2. Coordinate Reference Systems

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

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

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

Geographic coordinate reference systems

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

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

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

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

Query

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

Table 63. Result
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:

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

<table>
<thead>
<tr>
<th>Table 64. Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>bad</td>
</tr>
<tr>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

**8.10.3. Spatial instants**

Creating points

All point types are created from two components:

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

```sql
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 65. Result**

<table>
<thead>
<tr>
<th>cartesian_2d</th>
<th>cartesian_3d</th>
<th>geo_2d</th>
<th>geo_3d</th>
</tr>
</thead>
<tbody>
<tr>
<td>point({x: 3.0, y: 0.0, crs: 'cartesian'})</td>
<td>point({x: 0.0, y: 4.0, z: 1.0, crs: 'cartesian-3d'})</td>
<td>point({x: 56.0, y: 12.0, crs: 'wgs-84'})</td>
<td>point({x: 56.0, y: 12.0, z: 1000.0, crs: 'wgs-84-3d'})</td>
</tr>
</tbody>
</table>

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 66. Components of point instances and where they are supported**

<table>
<thead>
<tr>
<th>cartesian_2d</th>
<th>cartesian_3d</th>
<th>geo_2d</th>
<th>geo_3d</th>
</tr>
</thead>
<tbody>
<tr>
<td>point({x: 3.0, y: 0.0, crs: 'cartesian'})</td>
<td>point({x: 0.0, y: 4.0, z: 1.0, crs: 'cartesian-3d'})</td>
<td>point({x: 56.0, y: 12.0, crs: 'wgs-84'})</td>
<td>point({x: 56.0, y: 12.0, z: 1000.0, crs: 'wgs-84-3d'})</td>
</tr>
</tbody>
</table>

Rows: 1
<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
<th>Type</th>
<th>Range/Format</th>
<th>WGS-84</th>
<th>WGS-84-3D</th>
<th>Cartesian</th>
<th>Cartesian-3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>instant.x</td>
<td>The first element of the Coordinate</td>
<td>Float</td>
<td>Number literal, range depends on CRS</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>instant.y</td>
<td>The second element of the Coordinate</td>
<td>Float</td>
<td>Number literal, range depends on CRS</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>instant.z</td>
<td>The third element of the Coordinate</td>
<td>Float</td>
<td>Number literal, range depends on CRS</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>instant.latitude</td>
<td>The second element of the Coordinate for geographic CRS, degrees North of the equator</td>
<td>Float</td>
<td>Number literal, -90.0 to 90.0</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>instant.longitude</td>
<td>The first element of the Coordinate for geographic CRS, degrees East of the prime meridian</td>
<td>Float</td>
<td>Number literal, -180.0 to 180.0</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>instant.height</td>
<td>The third element of the Coordinate for geographic CRS, meters above the ellipsoid defined by the datum (WGS-84)</td>
<td>Float</td>
<td>Number literal, range limited only by the underlying 64-bit floating point type</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instant.crs</td>
<td>The name of the CRS</td>
<td>String</td>
<td>One of wgs-84, wgs-84-3d, cartesian, cartesian-3d</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>instant.srid</td>
<td>The internal Neo4j ID for the CRS</td>
<td>Integer</td>
<td>One of 4326, 4979, 7203, 9157</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

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

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

8.10.5. Comparability and Orderability

Points with different CRS are not comparable. This means that any function operating on two points of different types will return null. This is true of the distance function as well as inequality comparisons. If these are used in a predicate, they will cause the associated MATCH to return no results.
However, all types are orderable. The Point types will be ordered after Numbers and before Temporal types. Points with different CRS with be ordered by their SRID numbers. For the current set of four CRS, this means the order is WGS84, WGS84-3D, Cartesian, Cartesian-3D.

8.11. Lists

Cypher has comprehensive support for lists.

- Lists in general
- List comprehension
- Pattern comprehension
Information regarding operators such as list concatenation (+), element existence checking (IN) and access ([]) can be found here. The behavior of the IN and [] operators with respect to null is detailed here.

8.11.1. Lists in general

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

Query

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

Table 71. Result

<table>
<thead>
<tr>
<th>list</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[0,1,2,3,4,5,6,7,8,9]</td>
<td></td>
</tr>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

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

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

Query

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

Table 72. Result

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

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

Query

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

Table 73. Result

<table>
<thead>
<tr>
<th>range(0, 10)[-3]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

Finally, you can use ranges inside the brackets to return ranges of the list.
Out-of-bound slices are simply truncated, but out-of-bound single elements return null.
8.11.2. List comprehension

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

Query

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

Table 81. Result

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

Rows: 1

Either the WHERE part, or the expression, can be omitted, if you only want to filter or map respectively.

Query

```
RETURN [x IN range(0, 10) WHERE x % 2 = 0 ] AS result
```
8.11.3. Pattern comprehension

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

The following graph is used for the example below:

![Graph](image.png)

Graph

Query

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

Table 84. Result

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

Rows: 1

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

8.12. Maps

This section describes how to use maps in Cyphers.

- Literal maps
- Map projection
**Examples of map projection**

The following graph is used for the examples below:

![Graph](image)

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

### 8.12.1. Literal maps

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

**Query**

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

**Table 85. Result**

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

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

Rows: 1

### 8.12.2. Map projection

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

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

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

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

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

- All-properties selector - projects all key-value pairs from the `map_variable` value.

The following conditions apply:

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

Examples of map projections

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

Query

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

Table 86. Result

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

Rows: 1

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

Query

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

Table 87. Result

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

Rows: 2

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

Query

```cypher
MATCH (actor:Person {name: 'Charlie Sheen'})
RETURN actor.*, actor.age
```
Table 88. Result

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

Rows: 1

8.13. Working with null

This section describes working with the null value.

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

8.13.1. Introduction to null in Cypher

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

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

8.13.2. Logical operations with null

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

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

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a AND b</th>
<th>a OR b</th>
<th>a XOR b</th>
<th>NOT a</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>null</td>
<td>false</td>
<td>null</td>
<td>null</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>null</td>
<td>null</td>
<td>true</td>
<td>null</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>null</td>
<td>false</td>
<td>false</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>
8.13.3. The **IN** operator and **null**

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

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

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

8.13.4. The **[]** operator and **null**

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

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

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

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

8.13.5. Expressions that return **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)`

[2] This is in accordance with the Gregorian calendar; i.e. years AD/CE start at year 1, and the year before that (year 1 BC/BCE) is 0, while year 2 BCE is -1 etc.

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

[4] For dates from December 29, this could be the next year, and for dates until January 3 this could be the previous year, depending on how week 1 begins.

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

[6] For the nanosecond part of the epoch offset, the regular nanosecond component (`instant.nanosecond`) can be used.
Chapter 9. Clauses

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

- Reading clauses
- Projecting clauses
- Reading sub-clauses
- Reading hints
- Writing clauses
- Reading/Writing clauses
- Set operations
- Subquery clauses
- Multiple graphs
- Importing data
- Administration clauses

Reading clauses

These comprise clauses that read data from the database.

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

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

Projecting clauses

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

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

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

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

**Reading hints**

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

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

**Writing clauses**

These comprise clauses that write the data to the database.

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

**Reading/Writing clauses**

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

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MERGE</td>
<td>Ensures that a pattern exists in the graph. Either the pattern already exists, or it needs to be created.</td>
</tr>
<tr>
<td></td>
<td><strong>--- ON CREATE</strong></td>
</tr>
<tr>
<td></td>
<td><strong>--- ON MATCH</strong></td>
</tr>
<tr>
<td>CALL ... [YIELD ... ]</td>
<td>Invokes a procedure deployed in the database and return any results.</td>
</tr>
</tbody>
</table>

**Set operations**

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

**Subquery clauses**

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL { ... }</td>
<td>Evaluates a subquery, typically used for post-union processing or aggregations.</td>
</tr>
</tbody>
</table>

**Multiple graphs**

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

**Importing data**
**LOAD CSV**
Use when importing data from CSV files.

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

### Administration clauses

These comprise clauses used to manage databases, schema and security; further details can be found in Administration.

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE</td>
<td>Drop</td>
</tr>
<tr>
<td>CREATE</td>
<td>DROP INDEX</td>
</tr>
<tr>
<td>CREATE</td>
<td>DROP CONSTRAINT</td>
</tr>
<tr>
<td>Users, roles, privileges</td>
<td>Manage users, roles and privileges for database, graph and sub-graph access control.</td>
</tr>
</tbody>
</table>

## 9.1. MATCH

The **MATCH** clause is used to search for the pattern described in it.

- **Introduction**
- **Basic node finding**
  - Get all nodes
  - Get all nodes with a label
  - Related nodes
  - Match with labels
- **Relationship basics**
  - Outgoing relationships
  - Directed relationships and variable
  - Match on relationship type
  - Match on multiple relationship types
  - Match on relationship type and use a variable
- **Relationships in depth**
  - Relationship types with uncommon characters
  - Multiple relationships
9.1.1. Introduction

The \texttt{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 \textit{Patterns}.

\texttt{MATCH} is often coupled to a \texttt{WHERE} part which adds restrictions, or predicates, to the \texttt{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 \texttt{WHERE} should always be put together with the \texttt{MATCH} clause it belongs to.

\texttt{MATCH} can occur at the beginning of the query or later, possibly after a \texttt{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 \texttt{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 \texttt{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 \texttt{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 \textit{Indexes for search performance}, and more about specifying hints to force Neo4j to solve a query in a specific way in \textit{Planner hints and the USING keyword}. 
To understand more about the patterns used in the MATCH clause, read Patterns.

The following graph is used for the examples below:

Graph

9.1.2. Basic node finding

Get all nodes

By just specifying a pattern with a single node and no labels, all nodes in the graph will be returned.

Query

```
MATCH (n)
RETURN n
```

Returns all the nodes in the database.

Table 89. Result

<table>
<thead>
<tr>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node[0]{name:&quot;Charlie Sheen&quot;}</td>
</tr>
<tr>
<td>Node[1]{name:&quot;Martin Sheen&quot;}</td>
</tr>
<tr>
<td>Node[2]{name:&quot;Michael Douglas&quot;}</td>
</tr>
<tr>
<td>Node[3]{name:&quot;Oliver Stone&quot;}</td>
</tr>
<tr>
<td>Node[4]{name:&quot;Rob Reiner&quot;}</td>
</tr>
<tr>
<td>Node[5]{title:&quot;Wall Street&quot;}</td>
</tr>
<tr>
<td>Node[6]{title:&quot;The American President&quot;}</td>
</tr>
</tbody>
</table>

Rows: 7

Get all nodes with a label

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

Query

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

Returns all the movies in the database.

Table 90. Result
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 91. Result

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

Rows: 1

Match with labels

To constrain your pattern with labels on nodes, you add it to your pattern nodes, using the label syntax.

Query

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

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

Table 92. Result

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

Rows: 1

9.1.3. Relationship basics

Outgoing relationships

When the direction of a relationship is of interest, it is shown by using --> or <--, like this:
Query

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

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

Table 93. Result

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

Rows: 1

Directed relationships and variable

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

Query

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

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

Table 94. Result

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

Rows: 1

Match on relationship type

When you know the relationship type you want to match on, you can specify it by using a colon together with the relationship type.

Query

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

Returns all actors that ACTED_IN 'Wall Street'.

Table 95. Result

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

To match on one of multiple types, you can specify this by chaining them together with the pipe symbol `|`.

**Query**

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

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

**Table 96. Result**

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

Rows: 4

Match on relationship type and use a variable

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

**Query**

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

Returns `ACTED_IN` roles for 'Wall Street'.

**Table 97. Result**

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

Rows: 3

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

Relationship types with uncommon characters

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

Query

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

Which leads to the following graph:

![Graph](image)

Returns a relationship type with spaces in it.

Table 98. Result

<table>
<thead>
<tr>
<th>type(r)</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>TYPE INCLUDING A SPACE</em></td>
</tr>
</tbody>
</table>

Rows: 1

Multiple relationships

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

Query

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

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

Table 99. Result
Variable length relationships

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

Query

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

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

Table 100. Result

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

Rows: 3

Variable length relationships with multiple relationship types

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

Query

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

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

Table 101. Result

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

Rows: 3
Relationship variable in variable length relationships

When the connection between two nodes is of variable length, the list of relationships comprising the connection can be returned using the following syntax:

Query

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

Returns a list of relationships.

Table 102. Result

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

Match with properties on a variable length path

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

Query

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

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

![Graph](image)

Query

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

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

Table 103. Result
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 104. Result

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

Named paths

If you want to return or filter on a path in your pattern graph, you can introduce a named path.

Query

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

Returns the two paths starting from 'Michael Douglas'

Table 105. Result

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

Rows: 2
Matching on a bound relationship

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

Query

```cypher
MATCH (a)-[r]- (b)
WHERE id(r) = 0
RETURN a, b
```

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

Table 106. Result

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

Rows: 2

9.1.5. Shortest path

Single shortest path

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

Query

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

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

Table 107. Result

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

Rows: 1
Single shortest path with predicates

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

Query

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

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

Table 108. Result

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

Rows: 1

All shortest paths

Finds all the shortest paths between two nodes.

Query

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

Finds the two shortest paths between 'Martin Sheen' and 'Michael Douglas'.

Table 109. Result

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

Rows: 2

9.1.6. Get node or relationship by id

Node by id

Searching for nodes by id can be done with the id() function in a predicate.
Neo4j reuses its internal ids when nodes and relationships are deleted. This means that applications using, and relying on internal Neo4j ids, are brittle or at risk of making mistakes. It is therefore recommended to rather use application-generated ids.

Query

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

The corresponding node is returned.

Table 110. Result

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

Relationship by id

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

This is not the recommended practice. See Node by id for more information on the use of Neo4j ids.

Query

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

The relationship with id 0 is returned.

Table 111. Result

<table>
<thead>
<tr>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>:ACTED_IN[0]{role:&quot;Bud Fox&quot;}</td>
</tr>
</tbody>
</table>

Multiple nodes by id

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

Query

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

This returns the nodes listed in the IN expression.

Table 112. Result
9.2. OPTIONAL MATCH

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

- Introduction
- Optional relationships
- Properties on optional elements
- Optional typed and named relationship

9.2.1. Introduction

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

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

To understand the patterns used in the **OPTIONAL MATCH** clause, read Patterns.

The following graph is used for the examples below:

Graph

9.2.2. Optional relationships

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

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

Returns null, since the node has no outgoing relationships.

Table 113. Result

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

Rows: 1

9.2.3. Properties on optional elements

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

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 114. Result

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

Rows: 1

9.2.4. Optional typed and named relationship

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

Query

```
MATCH (a:Movie {title: 'Wall Street'})
OPTIONAL MATCH (a)-[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 115. Result

<table>
<thead>
<tr>
<th>a.title</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Wall Street&quot;</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>
9.3. RETURN

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

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

9.3.1. Introduction

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

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

Graph

9.3.2. Return nodes

To return a node, list it in the `RETURN` statement.

Query

```
MATCH (n {name: 'B'})
RETURN n
```
The example will return the node.

Table 116. Result

<table>
<thead>
<tr>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node<a href="">1</a></td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

9.3.3. Return relationships

To return a relationship, just include it in the `RETURN` list.

Query

```sql
MATCH (n { name: 'A'})-[r:KNOWS]->(c)
RETURN r
```

The relationship is returned by the example.

Table 117. Result

<table>
<thead>
<tr>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>:KNOWS[0]{}</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

9.3.4. Return property

To return a property, use the dot separator, like this:

Query

```sql
MATCH (n { name: 'A'})
RETURN n.name
```

The value of the property `name` gets returned.

Table 118. Result

<table>
<thead>
<tr>
<th>n.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;A&quot;</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

9.3.5. Return all elements

When you want to return all nodes, relationships and paths found in a query, you can use the `*` symbol.
Query

```csharp
MATCH p = (a {name: 'A'})-[r]->(b)
RETURN *
```

This returns the two nodes, the relationship and the path used in the query.

Table 119. Result

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

Rows: 2

9.3.6. Variable with uncommon characters

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

Query

```csharp
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 120. Result

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

Rows: 1

9.3.7. Column alias

If the name of the column should be different from the expression used, you can rename it by using AS <new name>.

Query

```csharp
MATCH (a {name: 'A'})
RETURN a.age AS SomethingTotallyDifferent
```

Returns the age property of a node, but renames the column.

Table 121. Result

<table>
<thead>
<tr>
<th>SomethingTotallyDifferent</th>
</tr>
</thead>
<tbody>
<tr>
<td>55</td>
</tr>
</tbody>
</table>
9.3.8. Optional properties

If a property might or might not be there, you can still select it as usual. It will be treated as `null` if it is missing.

Query

```
MATCH (n)
RETURN n.age
```

This example returns the age when the node has that property, or `null` if the property is not there.

Table 122. Result

<table>
<thead>
<tr>
<th>n.age</th>
</tr>
</thead>
<tbody>
<tr>
<td>55</td>
</tr>
</tbody>
</table>
| <null>

Rows: 2

9.3.9. Other expressions

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

Query

```
MATCH (a {name: 'A'})
RETURN a.age > 30, "I'm a literal", (a)--()
```

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

Table 123. Result

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

Rows: 1

9.3.10. Unique results

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

Query

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

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

Table 124. Result

| b  | Node[1](name: "B") | Rows: 1 |

9.4. WITH

The **WITH** clause allows query parts to be chained together, piping the results from one to be used as starting points or criteria in the next.

- Introduction
- Filter on aggregate function results
- Sort results before using collect on them
- Limit branching of a path search

9.4.1. Introduction

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

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

Another use is to filter on aggregated values. **WITH** is used to introduce aggregates which can then be used in predicates in **WHERE**. These aggregate expressions create new bindings in the results. **WITH** can also, like **RETURN**, alias expressions that are introduced into the results using the aliases as the binding name.

**WITH** is also used to separate reading from updating of the graph. Every part of a query must be either read-only or write-only. When going from a writing part to a reading part, the switch must be done with a **WITH** clause.

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.
Graph

9.4.2. Filter on aggregate function results

Aggregated results have to pass through a \texttt{WITH} clause to be able to filter on.

Query

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

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

Table 125. Result

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

Rows: 1

9.4.3. Sort results before using collect on them

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

Query

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

A list of the names of people in reverse order, limited to 3, is returned in a list.

Table 126. Result

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

114
9.4.4. Limit branching of a path search

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

Query

```cypher
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 127. Result

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

Rows: 2

9.5. UNWIND

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

- Introduction
- Unwinding a list
- Creating a distinct list
- Using **UNWIND** with any expression returning a list
- Using **UNWIND** with a list of lists
- Using **UNWIND** with an empty list
- Using **UNWIND** with an expression that is not a list
- Creating nodes from a list parameter

9.5.1. Introduction

With **UNWIND**, you can transform any list back into individual rows. These lists can be parameters that were passed in, previously **collect** -ed result or other list expressions.
One common usage of unwind is to create distinct lists. Another is to create data from parameter lists that are provided to the query.

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

### 9.5.2. Unwinding a list

We want to transform the literal list into rows named \(x\) and return them.

**Query**

```mermaid
diagram flow
   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 128. Result**

<table>
<thead>
<tr>
<th>(x)</th>
<th>(y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;val&quot;</td>
</tr>
<tr>
<td>2</td>
<td>&quot;val&quot;</td>
</tr>
<tr>
<td>3</td>
<td>&quot;val&quot;</td>
</tr>
<tr>
<td>&lt;null&gt;</td>
<td>&quot;val&quot;</td>
</tr>
</tbody>
</table>

Rows: 4

### 9.5.3. Creating a distinct list

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

**Query**

```mermaid
diagram flow
   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 129. Result**

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

Rows: 1

### 9.5.4. Using **UNWIND** with any expression returning a list

Any expression that returns a list may be used with **UNWIND**.
The two lists — a and b — are concatenated to form a new list, which is then operated upon by UNWIND.

Table 130. Result

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

9.5.5. Using UNWIND with a list of lists

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

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 131. Result

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

9.5.6. Using UNWIND with an empty list

Using an empty list with UNWIND will produce no rows, irrespective of whether or not any rows existed beforehand, or whether or not other values are being projected.
Essentially, \texttt{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 \texttt{UNWIND v}, where \texttt{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 \texttt{MATCH} that has no results.

Query

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

Table 132. Result

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

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

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

9.5.7. Using \texttt{UNWIND} with an expression that is not a list

Using \texttt{UNWIND} on an expression that does not return a list, will return the same result as using \texttt{UNWIND} on a list that just contains that expression. As an example, \texttt{UNWIND 5} is effectively equivalent to \texttt{UNWIND[5]}. The exception to this is when the expression returns \texttt{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 133. Result

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

9.5.8. Creating nodes from a list parameter

Create a number of nodes and relationships from a parameter-list without using \texttt{FOREACH}. 
Parameters

```
{
  "events" : [
    {
      "year" : 2014,
      "id" : 1
    },
    {
      "year" : 2014,
      "id" : 2
    }
  ]
}
```

Query

```
UNWIND $events AS event
MERGE (y:Year {year: event.year})
MERGE (y)<-[i: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 134. Result

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

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

9.6. WHERE

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

- Introduction
- Basic usage
  - Boolean operations
  - Filter on node label
  - Filter on node property
  - Filter on relationship property
  - Filter on dynamically-computed property
  - Property existence checking
- String matching
  - Prefix string search using `STARTS WITH`
9.6.1. Introduction

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

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

For **MATCH** and **OPTIONAL MATCH** on the other hand, **WHERE** adds constraints to the patterns described. It should not be seen as a filter after the matching is finished.

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

The following graph is used for the examples below:

![Graph](image)

**9.6.2. Basic usage**

**Boolean operations**

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

**Query**

```sql
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, n.age
```

**Table 135. Result**

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

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 136. Result

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

Rows: 1

Filter on node property

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

Query

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

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

Table 137. Result

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

Rows: 1

Filter on relationship property

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

Query

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

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

Table 138. Result

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

Rows: 1
Filter on dynamically-computed node property

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

**Query**

```sql
WITH 'AGE' AS proppname
MATCH (n:Person)
WHERE n[toLower(proppname)] < 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 139. Result**

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

Rows: 1

Property existence checking

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

**Query**

```sql
MATCH (n:Person)
WHERE exists(n.belt)
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 140. Result**

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

Rows: 1

9.6.3. String matching

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

Prefix string search using `STARTS WITH`

The `STARTS WITH` operator is used to perform case-sensitive matching on the beginning of a string.
The name and age for the 'Peter' node are returned because his name starts with 'Pet'.

Table 141. Result

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

Suffix string search using **ENDS WITH**

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

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

Table 142. Result

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

Substring search using **CONTAINS**

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

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

Table 143. Result

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

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

**Query**

```cypher
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 144. Result**

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

Rows: 1

### 9.6.4. Regular expressions

Cypher supports filtering using regular expressions. The regular expression syntax is inherited from the [Java regular expressions](https://docs.oracle.com/javase/7/docs/api/java/util/regex/package-summary.html). This includes support for flags that change how strings are matched, including case-insensitive (\(?i\)), multiline (\(?m\)) and dotall (\(?s\)). Flags are given at the beginning of the regular expression, for example `MATCH (n) WHERE n.name =~ '(?i)Lon.*' RETURN n` will return nodes with name 'London' or with name 'LonDoN'.

**Matching using regular expressions**

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

**Query**

```cypher
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 145. Result**

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

Rows: 1

**Escaping in regular expressions**

Characters like . or * have special meaning in a regular expression. To use these as ordinary characters, without special meaning, escape them.
The name, age and email for the 'Peter' node are returned because his email ends with '.com'.

Table 146. Result

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

Rows: 1

Case-insensitive regular expressions

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

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

Table 147. Result

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

Rows: 1

9.6.5. Using path patterns in WHERE

Filter on patterns

Patterns are expressions in Cypher, expressions that return a list of paths. List expressions are also predicates — an empty list represents false, and a non-empty represents true.

So, patterns are not only expressions, they are also predicates. The only limitation to your pattern is that you must be able to express it in a single path. You cannot use commas between multiple paths like you do in MATCH. You can achieve the same effect by combining multiple patterns with AND.

Note that you cannot introduce new variables here. Although it might look very similar to the MATCH patterns, the WHERE clause is all about eliminating matched paths. MATCH (a)-[*]->(b) is very different from WHERE (a)-[*]->(b). The first will produce a path for every path it can find between a and b, whereas the latter will eliminate any matched paths where a and b do not have a directed relationship chain between them.
Query

```
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 148. Result

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

Rows: 1

Filter on patterns using **NOT**

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

Query

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

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

Table 149. Result

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

Rows: 2

Filter on patterns with properties

You can also add properties to your patterns:

Query

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

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

Table 150. Result

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

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

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

Syntax:

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

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

9.6.6. Using existential subqueries in `WHERE`
Simple existential subquery

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

Query

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

Table 152. Result

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Andy&quot;</td>
</tr>
<tr>
<td>&quot;Peter&quot;</td>
</tr>
<tr>
<td>Rows: 2</td>
</tr>
</tbody>
</table>

Existential subquery with WHERE clause

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

Query

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

Table 153. Result

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Andy&quot;</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

Nesting existential subqueries

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

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

Table 154. Result

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

Rows: 1

9.6.7. 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 155. Result

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

Rows: 2

9.6.8. Missing properties and values

Default to **false** if property is missing

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

Query

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

Rows: 1

Default to true if property is missing

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

```sql
MATCH (n:Person)
WHERE n.belt = 'white' OR n.belt IS NULL
RETURN n.name, n.age, n.belt
ORDER BY n.name
```

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

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

Rows: 3

Filter on null

Sometimes you might want to test if a value or a variable is null. This is done just like SQL does it, using IS NULL. Also like SQL, the negative is IS NOT NULL, although NOT(IS NULL x) also works.

```sql
MATCH (person:Person)
WHERE person.name = 'Peter' AND person.belt IS NULL
RETURN person.name, person.age, person.belt
```

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

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

Rows: 1
9.6.9. Using ranges

Simple range

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

Query

```sql
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 159. Result

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

Rows: 2

Composite range

Several inequalities can be used to construct a range.

Query

```sql
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 160. Result

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

Rows: 1

9.7. ORDER BY

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

- Introduction
- Order nodes by property
9.7.1. Introduction

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 The use of indexes.

Graph

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

9.7.2. Order nodes by property

ORDER BY is used to sort the output.
The nodes are returned, sorted by their name.

**Table 161. Result**

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

Rows: 3

### 9.7.3. Order nodes by multiple properties

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

Query

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

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

**Table 162. Result**

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

Rows: 3

### 9.7.4. Order nodes by id

`ORDER BY` is used to sort the output.

Query

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

The nodes are returned, sorted by their internal id.
Table 163. Result

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

Rows: 3

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

9.7.5. Order nodes by expression

`ORDER BY` is used to sort the output.

Query

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

The nodes are returned, sorted by their properties.

Table 164. Result

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

Rows: 3

9.7.6. Order nodes in descending order

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

Query

```cypher
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 165. Result
9.7.7. Ordering `null`

When sorting the result set, `null` will always come at the end of the result set for ascending sorting, and first when doing descending sort.

**Query**

```sql
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 166. Result**

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

Rows: 3

9.8. SKIP

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

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

9.8.1. Introduction

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

9.8.2. Skip first three rows

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

Query

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

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

Table 167. Result

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

Rows: 2

9.8.3. Return middle two rows

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

Query

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

Two nodes from the middle are returned.

Table 168. Result

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

Rows: 2
9.8.4. Using an expression with **SKIP** to return a subset of the rows

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

**Query**

```sql
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 169. Result**

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

Rows: 4

9.9. **LIMIT**

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

- Introduction
- Return a subset of the rows
- Using an expression with **LIMIT** to return a subset of the rows

9.9.1. Introduction

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

**Graph**
9.9.2. Return a limited subset of the rows

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

**Query**

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

Limit to 3 rows by the example query.

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

Rows: 3

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

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

**Query**

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

Rows: 1

9.10. CREATE

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

- Create nodes
  - Create single node
  - Create multiple nodes
Create a node with a label
Create a node with multiple labels
Create node and add labels and properties
Return created node

Create relationships
Create a relationship between two nodes
Create a relationship and set properties

Create a full path

Use parameters with CREATE
Create node with a parameter for the properties
Create multiple nodes with a parameter for their properties

In the CREATE clause, patterns are used extensively. Read Patterns for an introduction.

9.10.1. Create nodes

Create single node

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

Query

```
CREATE (n)
```

Table 172. 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 173. 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 174. 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 175. 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 176. 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 177. Result

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

Rows: 1
Nodes created: 1
Properties set: 1

9.10.2. Create relationships

Create a relationship between two nodes

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

Query

```
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 178. Result

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

Rows: 1
Relationships created: 1

Create a relationship and set properties

Setting properties on relationships is done in a similar manner to how it’s done when creating nodes. Note that the values can be any expression.

Query

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

142
The type and name of the newly-created relationship is returned by the example query.

Table 179. Result

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

Rows: 1  
Relationships created: 1  
Properties set: 1

9.10.3. Create a full path

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

Query

```
CREATE p = (andy {'name': 'Andy'})-[[:WORKS_AT]]->(neo)<-[:WORKS_AT]-(michael {'name': 'Michael'})
RETURN p
```

This query creates three nodes and two relationships in one go, assigns it to a path variable, and returns it.

Table 180. Result

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

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

9.10.4. Use parameters with CREATE

Create node with a parameter for the properties

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

Parameters

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

Query

```
CREATE (n:Person $props)
RETURN n
```
Create multiple nodes with a parameter for their properties

By providing Cypher an array of maps, it will create a node for each map.

**Parameters**

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

**Query**

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

9.11. DELETE

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

- **Introduction**
- **Delete a single node**
- **Delete all nodes and relationships**
- **Delete a node with all its relationships**
- **Delete relationships only**
9.11.1. Introduction

For removing properties and labels, see REMOVE. Remember that you cannot delete a node without also deleting relationships that start or end on said node. Either explicitly delete the relationships, or use DETACH DELETE.

The examples start out with the following database:

![Database Diagram]

Graph

9.11.2. Delete single node

To delete a node, use the DELETE clause.

Query

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

Table 183. Result

( empty result )

Rows: 0
Nodes deleted: 1

9.11.3. Delete all nodes and relationships

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

Query

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

Table 184. Result

( empty result )

Rows: 0
Nodes deleted: 4
Relationships deleted: 2
9.11.4. Delete a node with all its relationships

When you want to delete a node and any relationship going to or from it, use **`DETACH DELETE`**.

**Query**

```
MATCH (n {name: 'Andy'})
DETACH DELETE n
```

**Table 185. Result**

<table>
<thead>
<tr>
<th>(empty result)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 0</td>
</tr>
<tr>
<td>Nodes deleted: 1</td>
</tr>
<tr>
<td>Relationships deleted: 2</td>
</tr>
</tbody>
</table>

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

9.11.5. Delete relationships only

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

**Query**

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

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

**Table 186. Result**

<table>
<thead>
<tr>
<th>(empty result)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 0</td>
</tr>
<tr>
<td>Relationships deleted: 2</td>
</tr>
</tbody>
</table>

9.12. **SET**

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

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

9.12.1. Introduction

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

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

The examples use this graph as a starting point:

Graph

9.12.2. Set a property

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

Query

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

The newly-changed node is returned by the query.

Table 187. Result
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 188. Result

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

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 189. Result

<table>
<thead>
<tr>
<th>n.name</th>
<th>n.worksIn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andy</td>
<td>null</td>
</tr>
</tbody>
</table>

9.12.3. Update a property

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

**Query**

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

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

### Table 190. Result

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

Rows: 1  
Properties set: 1

### 9.12.4. Remove a property

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

**Query**

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

The `name` property is now missing.

**Table 191. Result**

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

Rows: 1  
Properties set: 1

### 9.12.5. Copy properties between nodes and relationships

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

**Query**

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

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

**Table 192. Result**

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

Rows: 1  
Properties set: 3
9.12.6. Replace all properties using a map and =

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

**Query**

```cypher
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 193. Result**

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

Rows: 1
Properties set: 3

9.12.7. Remove all properties using an empty map and =

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

**Query**

```cypher
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 194. Result**

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

Rows: 1
Properties set: 2

9.12.8. Mutate specific properties using a map and +=

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

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

Query

```
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 195. Result

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

Rows: 1
Properties set: 3

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

Query

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

Table 196. Result

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

Rows: 1

9.12.9. Set multiple properties using one `SET` clause

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

Query

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

Table 197. Result

(Empty result)

Rows: 0
Properties set: 2
9.12.10. 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>
<thead>
<tr>
<th>n.name</th>
<th>n.surname</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Andy&quot;</td>
<td>&quot;Taylor&quot;</td>
</tr>
</tbody>
</table>

Rows: 1
Properties set: 1

9.12.11. Set all properties using a parameter

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

Parameters

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

Rows: 1
Properties set: 4
9.12.12. Set a label on a node

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

**Query**

```cypher
MATCH (n {name: 'Stefan'})
SET n:German
RETURN n.name, labels(n) AS labels
```

The newly-labeled node is returned by the query.

**Table 200. Result**

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

Rows: 1
Labels added: 1

9.12.13. Set multiple labels on a node

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

**Query**

```cypher
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 201. Result**

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

Rows: 1
Labels added: 2

9.13. REMOVE

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

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

9.13.1. Introduction

For deleting nodes and relationships, see DELETE.

Removing labels from a node is an idempotent operation: if you try to remove a label from a node that does not have that label on it, nothing happens. The query statistics will tell you if something needed to be done or not.

The examples use the following database:

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

Table 202. Result

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

Rows: 1
Properties set: 1

9.13.2. Remove a property

Neo4j doesn’t allow storing null in properties. Instead, if no value exists, the property is just not there. So, REMOVE is used to remove a property value from a node or a relationship.

Query

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

9.13.3. Remove all properties

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

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

**Query**

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

**Table 203. Result**

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

Rows: 1
Labels removed: 1

9.13.5. Remove multiple labels from a node

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

**Query**

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

**Table 204. Result**

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

Rows: 1
Labels removed: 2

9.14. **FOREACH**

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

9.14.1. Introduction

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

The variable context within the **FOREACH** parenthesis is separate from the one outside it. This means that if you **CREATE** a node variable within a **FOREACH**, you will not be able to use it outside of the foreach statement, unless you match to find it.
Within the `FOREACH` parentheses, you can do any of the updating commands — `SET`, `REMOVE`, `CREATE`, `MERGE`, `DELETE`, and `FOREACH`.

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

![Graph](image)

**9.14.2. Mark all nodes along a path**

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

**Query**

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

**Table 205. Result**

<table>
<thead>
<tr>
<th>Rows: 0</th>
<th>Properties set: 4</th>
</tr>
</thead>
</table>

**9.15. MERGE**

The `MERGE` clause ensures that a pattern exists in the graph. Either the pattern already exists, or it needs to be created.

- **Introduction**
- **Merge nodes**
  - Merge single node with a label
  - Merge single node with properties
• Merge single node specifying both label and property
• Merge single node derived from an existing node property

• Use ON CREATE and ON MATCH
  • Merge with ON CREATE
  • Merge with ON MATCH
  • Merge with ON CREATE and ON MATCH
  • Merge with ON MATCH setting multiple properties

• Merge relationships
  • Merge on a relationship
  • Merge on multiple relationships
  • Merge on an undirected relationship
  • Merge on a relationship between two existing nodes
  • Merge on a relationship between an existing node and a merged node derived from a node property

• Using unique constraints with MERGE
  • Merge using unique constraints creates a new node if no node is found
  • Merge using unique constraints matches an existing node
  • Merge with unique constraints and partial matches
  • Merge with unique constraints and conflicting matches

• Using map parameters with MERGE

9.15.1. Introduction

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

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

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

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

As with MATCH, MERGE can match multiple occurrences of a pattern. If there are multiple matches, they will all be passed on to later stages of the query.
The last part of **MERGE** is the **ON CREATE** and **ON MATCH**. These allow a query to express additional changes to the properties of a node or relationship, depending on if the element was matched (**MATCH**) in the database or if it was created (**CREATE**).

The following graph is used for the examples below:

9.15.2. Merge nodes

**Merge single node with a label**

Merging a single node with the given label.

**Query**

```sql
MERGE (robert:Critic)
RETURN robert, labels(robert)
```

A new node is created because there are no nodes labeled **Critic** in the database.

**Table 206. Result**

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

Rows: 1
Nodes created: 1
Labels added: 1

**Merge single node with properties**

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

**Query**

```sql
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 207. Result**

<table>
<thead>
<tr>
<th>charlie</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node[7]{name:&quot;Charlie Sheen&quot;,age:10}</td>
</tr>
</tbody>
</table>
Merge single node specifying both label and property

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

Query

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

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

Table 208. Result

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

Rows: 1

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

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

Merge single node derived from an existing node property

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

Query

```mermaid
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 209. Result

<table>
<thead>
<tr>
<th>person.name</th>
<th>person.bornIn</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Charlie Sheen&quot;</td>
<td>&quot;New York&quot;</td>
<td>Node[7]{name:&quot;New York&quot;}</td>
</tr>
<tr>
<td>&quot;Martin Sheen&quot;</td>
<td>&quot;Ohio&quot;</td>
<td>Node[8]{name:&quot;Ohio&quot;}</td>
</tr>
</tbody>
</table>
Rows: 5
Nodes created: 3
Properties set: 3
Labels added: 3

9.15.3. Use **ON CREATE** and **ON MATCH**

**Merge with **ON CREATE**

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

**Query**

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

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

**Table 210. Result**

<table>
<thead>
<tr>
<th>keanu.name</th>
<th>keanu.created</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Keanu Reeves&quot;</td>
<td>1660054736134</td>
</tr>
</tbody>
</table>

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

**Merge with **ON MATCH**

Merging nodes and setting properties on found nodes.

**Query**

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

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

**Table 211. Result**

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

Rows: 5
Properties set: 5

Merge with **ON CREATE** and **ON MATCH**

**Query**

```mermaid
diagram graph LR
  A[keanu:Person \(\text{name: 'Keanu Reeves'}\)]
  B[on create]
  C[set keanu.created = timestamp()]
  D[on match]
  E[set keanu.lastSeen = timestamp()]
  F[return keanu.name, keanu.created, keanu.lastSeen]
```

The query creates the 'keanu' node, and sets a timestamp on creation time. If 'keanu' had already existed, a different property would have been set.

**Table 212. Result**

<table>
<thead>
<tr>
<th>keanu.name</th>
<th>keanu.created</th>
<th>keanu.lastSeen</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Keanu Reeves&quot;</td>
<td>1660054738057</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

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

Merge with **ON MATCH** setting multiple properties

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

**Query**

```mermaid
diagram graph LR
  A[person:Person]
  B[on match]
  C[set person.found = true,]
  D[person.lastAccessed = timestamp()]
  E[return person.name, person.found, person.lastAccessed]
```

**Table 213. Result**

<table>
<thead>
<tr>
<th>person.name</th>
<th>person.found</th>
<th>person.lastAccessed</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Charlie Sheen&quot;</td>
<td>true</td>
<td>1660054739049</td>
</tr>
<tr>
<td>&quot;Martin Sheen&quot;</td>
<td>true</td>
<td>1660054739049</td>
</tr>
<tr>
<td>&quot;Michael Douglas&quot;</td>
<td>true</td>
<td>1660054739049</td>
</tr>
</tbody>
</table>
9.15.4. Merge relationships

Merge on a relationship

MERGE can be used to match or create a relationship.

Query

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

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

Table 214. Result

<table>
<thead>
<tr>
<th>charlie.name</th>
<th>type(r)</th>
<th>wallStreet.title</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Charlie Sheen&quot;</td>
<td>&quot;ACTED_IN&quot;</td>
<td>&quot;Wall Street&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

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 215. Result

<table>
<thead>
<tr>
<th>movie</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node[7]{}</td>
</tr>
</tbody>
</table>
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

```cypher
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 216. Result

<table>
<thead>
<tr>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>:KNOWS[8]{}</td>
</tr>
</tbody>
</table>

Rows: 1
Relationships created: 1

Merge on a relationship between two existing nodes

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

Query

```cypher
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 217. Result

<table>
<thead>
<tr>
<th>person.name</th>
<th>person.bornIn</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Charlie Sheen&quot;</td>
<td>&quot;New York&quot;</td>
<td>Node[7]{name:&quot;New York&quot;}</td>
</tr>
</tbody>
</table>
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)-[: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 218. Result**

<table>
<thead>
<tr>
<th>person.name</th>
<th>person.chauffeurName</th>
<th>chauffeur</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Charlie Sheen&quot;</td>
<td>&quot;John Brown&quot;</td>
<td>Node[7]{name:&quot;John Brown&quot;}</td>
</tr>
<tr>
<td>&quot;Martin Sheen&quot;</td>
<td>&quot;Bob Brown&quot;</td>
<td>Node[8]{name:&quot;Bob Brown&quot;}</td>
</tr>
<tr>
<td>&quot;Michael Douglas&quot;</td>
<td>&quot;John Brown&quot;</td>
<td>Node[9]{name:&quot;John Brown&quot;}</td>
</tr>
<tr>
<td>&quot;Oliver Stone&quot;</td>
<td>&quot;Bill White&quot;</td>
<td>Node[10]{name:&quot;Bill White&quot;}</td>
</tr>
<tr>
<td>&quot;Rob Reiner&quot;</td>
<td>&quot;Ted Green&quot;</td>
<td>Node[11]{name:&quot;Ted Green&quot;}</td>
</tr>
</tbody>
</table>

Rows: 5
Nodes created: 5
Relationships created: 5
Properties set: 5
Labels added: 5
9.15.5. Using unique constraints with `MERGE`

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

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

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

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

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

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>
<thead>
<tr>
<th>Table 219. Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>laurence.name</td>
</tr>
<tr>
<td>&quot;Laurence Fishburne&quot;</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
<tr>
<td>Nodes created: 1</td>
</tr>
<tr>
<td>Properties set: 1</td>
</tr>
<tr>
<td>Labels added: 1</td>
</tr>
</tbody>
</table>

Merge using unique constraints matches an existing node

Query

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

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

<table>
<thead>
<tr>
<th>Table 220. Result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
Merge with unique constraints and partial matches

Merge using unique constraints fails when finding partial matches.

Query

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

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

Error message

Merge did not find a matching node michael and can not create a new node due to conflicts with existing unique nodes

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

Query

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

Merge with unique constraints and conflicting matches

Merge using unique constraints fails when finding conflicting matches.

Query

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

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

Error message

Merge did not find a matching node oliver and can not create a new node due to conflicts with existing unique nodes

Using map parameters with MERGE

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

```json
{
  "param": {
    "name": "Keanu Reeves",
    "role": "Neo"
  }
}
```

Query

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

Table 221. Result

<table>
<thead>
<tr>
<th>person.name</th>
<th>person.role</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Keanu Reeves&quot;</td>
<td>&quot;Neo&quot;</td>
</tr>
</tbody>
</table>

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

9.16. CALL {} (subquery)

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

- Introduction
- Post-union processing
- Aggregation and side-effects
- Correlated subqueries

9.16.1. Introduction

CALL allows to execute subqueries, i.e. queries inside of other queries. Subqueries allow you to compose queries, which is especially useful when working with UNION or aggregations.

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

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

There are restrictions on what queries are allowed as subqueries and how they interact with the enclosing query:
• A subquery must end with a `RETURN` clause.
• A subquery cannot refer to variables from the enclosing query.
• A subquery cannot return variables with the same names as variables in the enclosing query.
• All variables that are returned from a subquery are afterwards available in the enclosing query.

The following graph is used for the examples below:

![Graph](image)

### 9.16.2. 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**

```sql
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 222. Result**

<table>
<thead>
<tr>
<th>p.name</th>
<th>p.age</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Alice&quot;</td>
<td>20</td>
</tr>
<tr>
<td>&quot;Charlie&quot;</td>
<td>65</td>
</tr>
</tbody>
</table>

Rows: 2

If different parts of a result should be matched differently, with some aggregation over the whole results, subqueries need to be used. This example query finds all persons with friends in one part of the union and all children with parents in the other part. Subsequently the number of friends and parents is counted together.
9.16.3. Aggregation and side-effects

Subqueries can be useful to do aggregations for each row and to isolate side-effects. This example query creates five Clone nodes for each existing person. The aggregation ensures that cardinality is not changed by the subquery. Without this, the result would be five times as many rows.

Query

```cypher
MATCH (p:Person)
CALL {
  UNWIND range(1, 5) AS i
  CREATE (c:Clone)
  RETURN count(c) AS numberOfClones
}
RETURN p.name, numberOfClones
```

Table 224. Result

<table>
<thead>
<tr>
<th>p.name</th>
<th>numberOfClones</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Alice&quot;</td>
<td>5</td>
</tr>
<tr>
<td>&quot;Bob&quot;</td>
<td>5</td>
</tr>
<tr>
<td>&quot;Charlie&quot;</td>
<td>5</td>
</tr>
<tr>
<td>&quot;Dora&quot;</td>
<td>5</td>
</tr>
</tbody>
</table>

Rows: 4

9.16.4. Correlated subqueries

A correlated subquery is a subquery that uses variables defined outside of the CALL clause. To be able to use a variable in this way, the variable must be explicitly imported into the subquery.

This functionality is currently only available in Neo4j Fabric, see Operations Manual → Fabric.
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.

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

Table 225. Result

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
</tr>
</tbody>
</table>

Rows: 3

An importing WITH clause must:

- Consist only of simple references to outside variables - e.g. WITH x, y, z. Aliasing or expressions are not supported in importing WITH clauses - e.g. WITH a AS b or WITH a + 1 AS b.
- Be the first clause of a subquery (or the second clause, if directly following a USE clause).

Aggregation on imported variables

Aggregations in subqueries are scoped to the subquery evaluation, also for imported variables, as shown in the following example:

Query.

```plaintext
UNWIND [0, 1, 2] AS x
CALL {
  WITH x
  RETURN max(x) AS xMax
}
RETURN x, xMax
```

Table 226. Result

<table>
<thead>
<tr>
<th>x</th>
<th>xMax</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Rows: 3
The aggregation $\text{max}(x)$ observes only a single value of $x$ in each evaluation of the subquery, and thus simply evaluates to that same value.

9.17. CALL procedure

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

9.17.1. Introduction

Procedures are called using the CALL clause.

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

Most procedures return a stream of records with a fixed set of result fields, similar to how running a Cypher query returns a stream of records. The YIELD sub-clause is used to explicitly select which of the available result fields are returned as newly-bound variables from the procedure call to the user or for further processing by the remaining query. Thus, in order to be able to use YIELD, the names (and types) of the output parameters need be known in advance. Each yielded result field may optionally be renamed using aliasing (i.e., `resultFieldName AS newName`). All new variables bound by a procedure call are added to the set of variables already bound in the current scope. It is an error if a procedure call tries to rebind a previously bound variable (i.e., a procedure call cannot shadow a variable that was previously bound in the current scope).

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.

9.17.2. Call a procedure using CALL

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

Query

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

Table 227. Result

<table>
<thead>
<tr>
<th>label</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;User&quot;</td>
</tr>
<tr>
<td>&quot;Administrator&quot;</td>
</tr>
</tbody>
</table>

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 228. Result

<table>
<thead>
<tr>
<th>label</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;User&quot;</td>
</tr>
<tr>
<td>&quot;Administrator&quot;</td>
</tr>
</tbody>
</table>

Rows: 2

9.17.3. View the signature for a procedure

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

Query

```
CALL dbms.procedures() YIELD name, signature
WHERE name='dbms.listConfig'
RETURN signature
```
We can see that the `dbms.listConfig` has one input parameter, `searchString`, and three output parameters, `name`, `description` and `value`.

Table 229. Result

<table>
<thead>
<tr>
<th>signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;dbms.listConfig(searchString = :: STRING?) :: (name :: STRING?, description :: STRING?, value :: STRING?, dynamic :: BOOLEAN?)&quot;</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

9.17.4. Call a procedure using a quoted namespace and name

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

Query

```sql
CALL 'db.labels()'
```

Query

```sql
CALL 'db.labels'
```

9.17.5. Call a procedure with literal arguments

This calls the example procedure `dbms.security.createUser` using literal arguments. The arguments are written out directly in the statement text.

Query

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

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

9.17.6. Call a procedure with parameter arguments

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

Examples that use parameter arguments shows the given parameters in JSON format; the exact manner in which they are to be submitted depends upon the driver being used. See Parameters, for more about querying with parameters

```json
{
  "username" : "example_username",
  "password" : "example_password",
  "requirePasswordChange" : false
}
```
Query

```
CALL dbms.security.createUser($username, $password, $requirePasswordChange)
```

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

Cypher allows the omission of parentheses for procedures with arity-n (n arguments), Cypher implicitly passes the parameter arguments.

Best practice is to use parentheses for procedures. Omission of parantheses is available only in a so-called standalone procedure call, when the whole query consists of a single CALL clause.

Parameters

```json
{
   "username" : "example_username",
   "password" : "example_password",
   "requirePasswordChange" : false
}
```

Query

```
CALL dbms.security.createUser
```

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

9.17.7. Call a procedure with mixed literal and parameter arguments

This calls the example procedure `dbms.security.createUser` using both literal and parameter arguments.

Parameters

```json
{
   "password" : "example_password"
}
```

Query

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

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

9.17.8. Call a procedure with literal and default arguments

This calls the example procedure `dbms.security.createUser` using literal arguments. That is, arguments that are written out directly in the statement text, and a trailing default argument that is provided by the procedure itself.
Since our example procedure does not return any result, the result is empty.

9.17.9. Call a procedure within a complex query using CALL YIELD

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

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

9.17.10. Call a procedure and filter its results

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

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

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

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

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

9.18. UNION

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

- Introduction
9.18.1. Introduction

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

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

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

Graph

9.18.2. Combine two queries and retain duplicates

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

Query

```
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 230. Result

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Anthony Hopkins&quot;</td>
</tr>
<tr>
<td>&quot;Helen Mirren&quot;</td>
</tr>
<tr>
<td>&quot;Hitchcock&quot;</td>
</tr>
<tr>
<td>&quot;Hitchcock&quot;</td>
</tr>
</tbody>
</table>

Rows: 4
9.18.3. Combine two queries and remove duplicates

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

**Query**

```cypher
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 231. Result**

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Anthony Hopkins&quot;</td>
</tr>
<tr>
<td>&quot;Helen Mirren&quot;</td>
</tr>
<tr>
<td>&quot;Hitchcock&quot;</td>
</tr>
</tbody>
</table>

Rows: 3

9.19. **USE Fabric**

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

- **Introduction**
- **Syntax**
- **Examples**
  - Query remote graph by name
  - Query remote graph by graph ID

9.19.1. **Introduction**

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

- This functionality is currently only available in Neo4j Fabric. Find out more about this feature in Operations Manual → Fabric.

9.19.2. **Syntax**

The **USE** clause can only appear as the first clause of:

- Queries:
• 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.

9.19.3. Examples

In the following examples we assume that we have configured a Fabric database called `exampleFabricSetup`.

**Query graph by name**

The graph that we wish to query is configured to be referred to by the name `exampleDatabaseName`.

```
USE exampleFabricSetup.exampleDatabaseName
```

**Query graph by graph ID**

The graph we wish to query is configured with the graph id `0`, which is why we can refer to it using the built-in function `graph()` with the argument `0`:

```
USE exampleFabricSetup.graph(0)
```

9.20. LOAD CSV

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

• Introduction
• CSV file format
• Import data from a CSV file
• Import data from a remote CSV file
• Import data from a CSV file containing headers
• Import data from a CSV file with a custom field delimiter
• Importing large amounts of data
• Setting the rate of periodic commits
• Import data containing escaped characters
• Using linenumber() with LOAD CSV
• Using file() with LOAD CSV

9.20.1. Introduction

• The URL of the CSV file is specified by using FROM followed by an arbitrary expression evaluating to the URL in question.
• It is required to specify a variable for the CSV data using AS.
• CSV files can be stored on the database server and are then accessible using a file:// URL. Alternatively, LOAD CSV also supports accessing CSV files via HTTPS, HTTP, and FTP.
• LOAD CSV supports resources compressed with gzip and Deflate. Additionally LOAD CSV supports locally stored CSV files compressed with ZIP.
• LOAD CSV will follow HTTP redirects but for security reasons it will not follow redirects that changes the protocol, for example if the redirect is going from HTTPS to HTTP.
• LOAD CSV is often used in conjunction with the query hint PERIODIC COMMIT; more information on this may be found in PERIODIC COMMIT query hint.

Configuration settings for file URLs

dbms.security.allow_csv_import_from_file_urls

This setting determines if Cypher will allow the use of file:/// URLs when loading data using LOAD CSV. Such URLs identify files on the filesystem of the database server. Default is true. Setting dbms.security.allow_csv_import_from_file_urls=false will completely disable access to the file system for LOAD CSV.

dbmsdirectories.import

Sets the root directory for file:/// URLs used with the Cypher LOAD CSV clause. This should be set to a single directory relative to the Neo4j installation path on the database server. All requests to load from file:/// URLs will then be relative to the specified directory. The default value set in the config settings is import. This is a security measure which prevents the database from accessing files outside the standard import directory, similar to how a Unix chroot operates. Setting this to an empty field will allow access to all files within the Neo4j installation folder. Commenting out this setting will disable the security feature, allowing all files in the local system to be imported. This is definitely not recommended.

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

- When using file:/// URLs, spaces and other non-alphanumeric characters need to be URL encoded. 

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

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

The file location is relative to the import. The config setting dbms.directories.import only applies to local disc and not to remote URLs.

See the examples below for further details.

9.20.2. CSV file format

The CSV file to use with LOAD CSV must have the following characteristics:

- the character encoding is UTF-8;
- the end line termination is system dependent, e.g., it is \n on unix or \r\n on windows;
- the default field terminator is ,;
- the field terminator character can be change by using the option FIELDTERMINATOR available in the LOAD CSV command;
- quoted strings are allowed in the CSV file and the quotes are dropped when reading the data;
- the character for string quotation is double quote ";
- if dbms.import.csv.legacy_quote_escaping is set to the default value of true, \ is used as an escape character;
- a double quote must be in a quoted string and escaped, either with the escape character or a second double quote.

9.20.3. Import data from a CSV file

To import data from a CSV file into Neo4j, you can use LOAD CSV to get the data into your query. Then you write it to your database using the normal updating clauses of Cypher.

artists.csv

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ABBA</td>
<td>1992</td>
</tr>
<tr>
<td>2</td>
<td>Roxette</td>
<td>1986</td>
</tr>
<tr>
<td>3</td>
<td>Europe</td>
<td>1979</td>
</tr>
<tr>
<td>4</td>
<td>The Cardigans</td>
<td>1992</td>
</tr>
</tbody>
</table>
A new node with the `Artist` label is created for each row in the CSV file. In addition, two columns from the CSV file are set as properties on the nodes.

**Result**

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

### 9.20.4. Import data from a remote CSV file

Accordingly, you can import data from a CSV file in a remote location into Neo4j. Note that this applies to all variations of CSV files (see examples below for other variations).

**data.neo4j.com/bands/artists.csv**

```
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
```

### 9.20.5. Import data from a CSV file containing headers

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

**artists-with-headers.csv**

```
Id, Name, Year
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
```
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.

<table>
<thead>
<tr>
<th>No data returned.</th>
</tr>
</thead>
</table>
Nodes created: 4
Properties set: 8
Labels added: 4

9.20.6. Import data from a CSV file with a custom field delimiter

Sometimes, your CSV file has other field delimiters than commas. You can specify which delimiter your file uses, using `FIELDTERMINATOR`. Hexadecimal representation of the unicode character encoding can be used if prepended by `\u`. The encoding must be written with four digits. For example, `\u003B` is equivalent to `;` (SEMICOLON).

```
artists-fieldterminator.csv
1;ABBA;1992
2;Roxette;1986
3;Europe;1979
4;The Cardigans;1992
```

As values in this file are separated by a semicolon, a custom `FIELDTERMINATOR` is specified in the `LOAD CSV` clause.

<table>
<thead>
<tr>
<th>No data returned.</th>
</tr>
</thead>
</table>
Nodes created: 4
Properties set: 8
Labels added: 4

9.20.7. Importing large amounts of data

If the CSV file contains a significant number of rows (approaching hundreds of thousands or millions), `USING PERIODIC COMMIT` can be used to instruct Neo4j to perform a commit after a number of rows. This reduces the memory overhead of the transaction state. By default, the commit will happen every 1000 rows. For more information, see `PERIODIC COMMIT` query hint.
9.20.8. Setting the rate of periodic commits

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

```
USING PERIODIC COMMIT 500 LOAD CSV FROM 'file:///artists.csv' AS line
CREATE (:Artist {name: line[1], year: toInteger(line[2])})
```

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

9.20.9. Import data containing escaped characters

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

```
"!","The "Symbol"","1992"
```

```
LOAD CSV FROM 'file:///artists-with-escaped-char.csv' AS line
CREATE (a:Artist {name: line[1], year: toInteger(line[2])})
RETURN 
a.name AS name,
a.year AS year,
size(a.name) AS size
```

Note that strings are wrapped in quotes in the output here. You can see that when comparing to the length of the string in this case!
9.20.10. Using linenumber() with LOAD CSV

For certain scenarios, like debugging a problem with a csv file, it may be useful to get the current line number that LOAD CSV is operating on. The linenumber() function provides exactly that or null if called without a LOAD CSV context.

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

9.20.11. Using file() with LOAD CSV

For certain scenarios, like debugging a problem with a csv file, it may be useful to get the absolute path of the file that LOAD CSV is operating on. The file() function provides exactly that or null if called without a LOAD CSV context.

artists.csv

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

```
LOAD CSV FROM 'file:///artists.csv' AS line
RETURN 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

<table>
<thead>
<tr>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>/home/example/neo4j/import/artists.csv</code></td>
</tr>
</tbody>
</table>

1 row

Chapter 10. Functions

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

- Predicate functions [Summary|Detail]
- Scalar functions [Summary|Detail]
- Aggregating functions [Summary|Detail]
- List functions [Summary|Detail]
- Mathematical functions - numeric [Summary|Detail]
- Mathematical functions - logarithmic [Summary|Detail]
- Mathematical functions - trigonometric [Summary|Detail]
- String functions [Summary|Detail]
- Temporal functions - instant types [Summary|Detail]
- Temporal functions - duration [Summary|Detail]
- Spatial functions [Summary|Detail]
- User-defined functions [Summary|Detail]
- LOAD CSV functions [Summary|Detail]

Related information may be found in Operators.

Please note
- Functions in Cypher return null if an input parameter is null.
- Functions taking a string as input all operate on Unicode characters rather than on a standard char[]. For example, the size() function applied to any Unicode character will return 1, even if the character does not fit in the 16 bits of one char.

Predicate functions

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

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

These functions return a single value.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>coalesce()</td>
<td>Returns the first non-null value in a list of expressions.</td>
</tr>
<tr>
<td>endNode()</td>
<td>Returns the end node of a relationship.</td>
</tr>
<tr>
<td>head()</td>
<td>Returns the first element in a list.</td>
</tr>
<tr>
<td>id()</td>
<td>Returns the id of a relationship or node.</td>
</tr>
<tr>
<td>last()</td>
<td>Returns the last element in a list.</td>
</tr>
<tr>
<td>length()</td>
<td>Returns the length of a path.</td>
</tr>
<tr>
<td>properties()</td>
<td>Returns a map containing all the properties of a node or relationship.</td>
</tr>
<tr>
<td>randomUUID()</td>
<td>Returns a string value corresponding to a randomly-generated UUID.</td>
</tr>
<tr>
<td>size()</td>
<td>Returns the number of items in a list.</td>
</tr>
<tr>
<td>size() applied to pattern expression</td>
<td>Returns the number of paths matching the pattern expression.</td>
</tr>
<tr>
<td>size() applied to string</td>
<td>Returns the number of Unicode characters in a string.</td>
</tr>
<tr>
<td>startNode()</td>
<td>Returns the start node of a relationship.</td>
</tr>
<tr>
<td>timestamp()</td>
<td>Returns the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC.</td>
</tr>
<tr>
<td>toBoolean()</td>
<td>Converts a string value to a boolean value.</td>
</tr>
<tr>
<td>toFloat()</td>
<td>Converts an integer or string value to a floating point number.</td>
</tr>
<tr>
<td>toInteger()</td>
<td>Converts a floating point or string value to an integer value.</td>
</tr>
<tr>
<td>type()</td>
<td>Returns the string representation of the relationship type.</td>
</tr>
</tbody>
</table>

### Aggregating functions

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg() - Numeric values</td>
<td>Returns the average of a set of numeric values.</td>
</tr>
<tr>
<td>avg() - Durations</td>
<td>Returns the average of a set of Durations.</td>
</tr>
<tr>
<td>collect()</td>
<td>Returns a list containing the values returned by an expression.</td>
</tr>
<tr>
<td>count()</td>
<td>Returns the number of values or rows.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>max()</td>
<td>Returns the maximum value in a set of values.</td>
</tr>
<tr>
<td>min()</td>
<td>Returns the minimum value in a set of values.</td>
</tr>
<tr>
<td>percentileCont()</td>
<td>Returns the percentile of a value over a group using linear interpolation.</td>
</tr>
<tr>
<td>percentileDisc()</td>
<td>Returns the nearest value to the given percentile over a group using a rounding method.</td>
</tr>
<tr>
<td>stDev()</td>
<td>Returns the standard deviation for the given value over a group for a sample of a population.</td>
</tr>
<tr>
<td>stDevP()</td>
<td>Returns the standard deviation for the given value over a group for an entire population.</td>
</tr>
<tr>
<td>sum() - Numeric values</td>
<td>Returns the sum of a set of numeric values.</td>
</tr>
<tr>
<td>sum() - Durations</td>
<td>Returns the sum of a set of Durations.</td>
</tr>
</tbody>
</table>

**List functions**

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>keys()</td>
<td>Returns a list containing the string representations for all the property names of a node, relationship, or map.</td>
</tr>
<tr>
<td>labels()</td>
<td>Returns a list containing the string representations for all the labels of a node.</td>
</tr>
<tr>
<td>nodes()</td>
<td>Returns a list containing all the nodes in a path.</td>
</tr>
<tr>
<td>range()</td>
<td>Returns a list comprising all integer values within a specified range.</td>
</tr>
<tr>
<td>reduce()</td>
<td>Runs an expression against individual elements of a list, storing the result of the expression in an accumulator.</td>
</tr>
<tr>
<td>relationships()</td>
<td>Returns a list containing all the relationships in a path.</td>
</tr>
<tr>
<td>reverse()</td>
<td>Returns a list in which the order of all elements in the original list have been reversed.</td>
</tr>
<tr>
<td>tail()</td>
<td>Returns all but the first element in a list.</td>
</tr>
</tbody>
</table>

**Mathematical functions - numeric**

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs()</td>
<td>Returns the absolute value of a number.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>ceil()</td>
<td>Returns the smallest floating point number that is greater than or equal to a number and equal to a mathematical integer.</td>
</tr>
<tr>
<td>floor()</td>
<td>Returns the largest floating point number that is less than or equal to a number and equal to a mathematical integer.</td>
</tr>
<tr>
<td>rand()</td>
<td>Returns a random floating point number in the range from 0 (inclusive) to 1 (exclusive); i.e. ([0,1)).</td>
</tr>
<tr>
<td>round()</td>
<td>Returns the value of a number rounded to the nearest integer.</td>
</tr>
<tr>
<td>sign()</td>
<td>Returns the signum of a number: 0 if the number is 0, -1 for any negative number, and 1 for any positive number.</td>
</tr>
</tbody>
</table>

**Mathematical functions - logarithmic**

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

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

**Mathematical functions - trigonometric**

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

All trigonometric functions operate on radians, unless otherwise specified.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acos()</td>
<td>Returns the arccosine of a number in radians.</td>
</tr>
<tr>
<td>asin()</td>
<td>Returns the arcsine of a number in radians.</td>
</tr>
<tr>
<td>atan()</td>
<td>Returns the arctangent of a number in radians.</td>
</tr>
<tr>
<td>atan2()</td>
<td>Returns the arctangent2 of a set of coordinates in radians.</td>
</tr>
<tr>
<td>cos()</td>
<td>Returns the cosine of a number.</td>
</tr>
<tr>
<td>cot()</td>
<td>Returns the cotangent of a number.</td>
</tr>
<tr>
<td>degrees()</td>
<td>Converts radians to degrees.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>haversin()</td>
<td>Returns half the versine of a number.</td>
</tr>
<tr>
<td>pi()</td>
<td>Returns the mathematical constant π.</td>
</tr>
<tr>
<td>radians()</td>
<td>Converts degrees to radians.</td>
</tr>
<tr>
<td>sin()</td>
<td>Returns the sine of a number.</td>
</tr>
<tr>
<td>tan()</td>
<td>Returns the tangent of a number.</td>
</tr>
</tbody>
</table>

**String functions**

These functions are used to manipulate strings or to create a string representation of another value.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>left()</td>
<td>Returns a string containing the specified number of leftmost characters of the original string.</td>
</tr>
<tr>
<td>lTrim()</td>
<td>Returns the original string with leading whitespace removed.</td>
</tr>
<tr>
<td>replace()</td>
<td>Returns a string in which all occurrences of a specified string in the original string have been replaced by another (specified) string.</td>
</tr>
<tr>
<td>reverse()</td>
<td>Returns a string in which the order of all characters in the original string have been reversed.</td>
</tr>
<tr>
<td>right()</td>
<td>Returns a string containing the specified number of rightmost characters of the original string.</td>
</tr>
<tr>
<td>rTrim()</td>
<td>Returns the original string with trailing whitespace removed.</td>
</tr>
<tr>
<td>split()</td>
<td>Returns a list of strings resulting from the splitting of the original string around matches of the given delimiter.</td>
</tr>
<tr>
<td>substring()</td>
<td>Returns a substring of the original string, beginning with a 0-based index start and length.</td>
</tr>
<tr>
<td>toLower()</td>
<td>Returns the original string in lowercase.</td>
</tr>
<tr>
<td>toString()</td>
<td>Converts an integer, float, boolean or temporal type (i.e. Date, Time, LocalTime, DateTime, LocalDateTime or Duration) value to a string.</td>
</tr>
<tr>
<td>toUpper()</td>
<td>Returns the original string in uppercase.</td>
</tr>
<tr>
<td>trim()</td>
<td>Returns the original string with leading and trailing whitespace removed.</td>
</tr>
</tbody>
</table>

**Temporal functions - instant types**

Values of the temporal types — Date, Time, LocalTime, DateTime, and LocalDateTime — can be created manipulated using the following functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date()</td>
<td>Returns the current Date.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>date.transaction()</td>
<td>Returns the current Date using the transaction clock.</td>
</tr>
<tr>
<td>date.statement()</td>
<td>Returns the current Date using the statement clock.</td>
</tr>
<tr>
<td>date.realtime()</td>
<td>Returns the current Date using the realtime clock.</td>
</tr>
<tr>
<td>date({year [, month, day]})</td>
<td>Returns a calendar (Year-Month-Day) Date.</td>
</tr>
<tr>
<td>date({year [, week, dayOfWeek]})</td>
<td>Returns a week (Year-Week-Day) Date.</td>
</tr>
<tr>
<td>date({year [, quarter, dayOfQuarter]})</td>
<td>Returns a quarter (Year-Quarter-Day) Date.</td>
</tr>
<tr>
<td>date({year [, ordinalDay]})</td>
<td>Returns an ordinal (Year-Day) Date.</td>
</tr>
<tr>
<td>date(string)</td>
<td>Returns a Date by parsing a string.</td>
</tr>
<tr>
<td>date({map})</td>
<td>Returns a Date from a map of another temporal value’s components.</td>
</tr>
<tr>
<td>date.truncate()</td>
<td>Returns a Date obtained by truncating a value at a specific component boundary. Truncation summary.</td>
</tr>
<tr>
<td>datetime()</td>
<td>Returns the current DateTime.</td>
</tr>
<tr>
<td>datetime.transaction()</td>
<td>Returns the current DateTime using the transaction clock.</td>
</tr>
<tr>
<td>datetime.statement()</td>
<td>Returns the current DateTime using the statement clock.</td>
</tr>
<tr>
<td>datetime.realtime()</td>
<td>Returns the current DateTime using the realtime clock.</td>
</tr>
<tr>
<td>datetime({year [, month, day, …]})</td>
<td>Returns a calendar (Year-Month-Day) DateTime.</td>
</tr>
<tr>
<td>datetime({year [, week, dayOfWeek, …]})</td>
<td>Returns a week (Year-Week-Day) DateTime.</td>
</tr>
<tr>
<td>datetime({year [, quarter, dayOfQuarter, …]})</td>
<td>Returns a quarter (Year-Quarter-Day) DateTime.</td>
</tr>
<tr>
<td>datetime({year [, ordinalDay, …]})</td>
<td>Returns an ordinal (Year-Day) DateTime.</td>
</tr>
<tr>
<td>datetime(string)</td>
<td>Returns a DateTime by parsing a string.</td>
</tr>
<tr>
<td>datetime({map})</td>
<td>Returns a DateTime from a map of another temporal value’s components.</td>
</tr>
<tr>
<td>datetime({epochSeconds})</td>
<td>Returns a DateTime from a timestamp.</td>
</tr>
<tr>
<td>datetime.truncate()</td>
<td>Returns a DateTime obtained by truncating a value at a specific component boundary. Truncation summary.</td>
</tr>
<tr>
<td>localdatetime()</td>
<td>Returns the current LocalDateTime.</td>
</tr>
<tr>
<td>localdatetime.transaction()</td>
<td>Returns the current LocalDateTime using the transaction clock.</td>
</tr>
<tr>
<td>localdatetime.statement()</td>
<td>Returns the current LocalDateTime using the statement clock.</td>
</tr>
<tr>
<td>localdatetime.realtime()</td>
<td>Returns the current LocalDateTime using the realtime clock.</td>
</tr>
<tr>
<td>localdatetime({year [, month, day, …]})</td>
<td>Returns a calendar (Year-Month-Day) LocalDateTime.</td>
</tr>
<tr>
<td>localdatetime({year [, week, dayOfWeek, …]})</td>
<td>Returns a week (Year-Week-Day) LocalDateTime.</td>
</tr>
<tr>
<td>localdatetime({year [, quarter, dayOfQuarter, …]})</td>
<td>Returns a quarter (Year-Quarter-Day) DateTime.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>localdatetime({year [, ordinalDay, ...])})</code></td>
<td>Returns an ordinal (Year-Day) LocalDateTime.</td>
</tr>
<tr>
<td><code>localdatetime(string)</code></td>
<td>Returns a LocalDateTime by parsing a string.</td>
</tr>
<tr>
<td><code>localdatetime({map})</code></td>
<td>Returns a LocalDateTime from a map of another temporal value's components.</td>
</tr>
<tr>
<td><code>localdatetime.truncate()</code></td>
<td>Returns a LocalDateTime obtained by truncating a value at a specific component boundary. Truncation summary.</td>
</tr>
<tr>
<td><code>localtime()</code></td>
<td>Returns the current LocalTime.</td>
</tr>
<tr>
<td><code>localtime.transaction()</code></td>
<td>Returns the current LocalTime using the transaction clock.</td>
</tr>
<tr>
<td><code>localtime.statement()</code></td>
<td>Returns the current LocalTime using the statement clock.</td>
</tr>
<tr>
<td><code>localtime.realtime()</code></td>
<td>Returns the current LocalTime using the realtime clock.</td>
</tr>
<tr>
<td><code>localtime({hour [, minute, second, ...])})</code></td>
<td>Returns a LocalTime with the specified component values.</td>
</tr>
<tr>
<td><code>localtime(string)</code></td>
<td>Returns a LocalTime by parsing a string.</td>
</tr>
<tr>
<td><code>localtime({time [, hour, ...])})</code></td>
<td>Returns a LocalTime from a map of another temporal value's components.</td>
</tr>
<tr>
<td><code>localtime.truncate()</code></td>
<td>Returns a LocalTime obtained by truncating a value at a specific component boundary. Truncation summary.</td>
</tr>
<tr>
<td><code>time()</code></td>
<td>Returns the current Time.</td>
</tr>
<tr>
<td><code>time.transaction()</code></td>
<td>Returns the current Time using the transaction clock.</td>
</tr>
<tr>
<td><code>time.statement()</code></td>
<td>Returns the current Time using the statement clock.</td>
</tr>
<tr>
<td><code>time.realtime()</code></td>
<td>Returns the current Time using the realtime clock.</td>
</tr>
<tr>
<td><code>time({hour [, minute, ...])})</code></td>
<td>Returns a Time with the specified component values.</td>
</tr>
<tr>
<td><code>time(string)</code></td>
<td>Returns a Time by parsing a string.</td>
</tr>
<tr>
<td><code>time({time [, hour, ..., timezone])})</code></td>
<td>Returns a Time from a map of another temporal value's components.</td>
</tr>
<tr>
<td><code>time.truncate()</code></td>
<td>Returns a Time obtained by truncating a value at a specific component boundary. Truncation summary.</td>
</tr>
</tbody>
</table>

**Temporal functions - duration**

Duration values of the temporal types can be created manipulated using the following functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>duration({map})</code></td>
<td>Returns a Duration from a map of its components.</td>
</tr>
<tr>
<td><code>duration(string)</code></td>
<td>Returns a Duration by parsing a string.</td>
</tr>
<tr>
<td><code>duration.between()</code></td>
<td>Returns a Duration equal to the difference between two given instants.</td>
</tr>
</tbody>
</table>
### Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>duration.inMonths()</td>
<td>Returns a <code>Duration</code> equal to the difference in whole months, quarters or years between two given instants.</td>
</tr>
<tr>
<td>duration.inDays()</td>
<td>Returns a <code>Duration</code> equal to the difference in whole days or weeks between two given instants.</td>
</tr>
<tr>
<td>duration.inSeconds()</td>
<td>Returns a <code>Duration</code> equal to the difference in seconds and fractions of seconds, or minutes or hours, between two given instants.</td>
</tr>
</tbody>
</table>

### Spatial functions

These functions are used to specify 2D or 3D points in a geographic or cartesian Coordinate Reference System and to calculate the geodesic distance between two points.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>distance()</td>
<td>Returns a floating point number representing the geodesic distance between any two points in the same CRS.</td>
</tr>
<tr>
<td>point() - Cartesian 2D</td>
<td>Returns a 2D point object, given two coordinate values in the Cartesian coordinate system.</td>
</tr>
<tr>
<td>point() - Cartesian 3D</td>
<td>Returns a 3D point object, given three coordinate values in the Cartesian coordinate system.</td>
</tr>
<tr>
<td>point() - WGS 84 2D</td>
<td>Returns a 2D point object, given two coordinate values in the WGS 84 geographic coordinate system.</td>
</tr>
<tr>
<td>point() - WGS 84 3D</td>
<td>Returns a 3D point object, given three coordinate values in the WGS 84 geographic coordinate system.</td>
</tr>
</tbody>
</table>

### User-defined functions

User-defined functions are written in Java, deployed into the database and are called in the same way as any other Cypher function. There are two main types of functions that can be developed and used:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Usage</th>
<th>Developing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar</td>
<td>For each row the function takes parameters and returns a result</td>
<td>Using UDF</td>
<td>Extending Neo4j (UDF)</td>
</tr>
<tr>
<td>Aggregating</td>
<td>Consumes many rows and produces an aggregated result</td>
<td>Using aggregating UDF</td>
<td>Extending Neo4j (Aggregating UDF)</td>
</tr>
</tbody>
</table>

### LOAD CSV functions

LOAD CSV functions can be used to get information about the file that is processed by `LOAD CSV`.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>linenumber()</td>
<td>Returns the line number that <code>LOAD CSV</code> is currently using.</td>
</tr>
</tbody>
</table>
10.1. Predicate functions

Predicates are boolean functions that return true or false for a given set of non-null input. They are most commonly used to filter out paths in the WHERE part of a query.

Functions:
- all()
- any()
- exists()
- none()
- single()

Graph

10.1.1. all()

The function all() returns true if the predicate holds for all elements in the given list. null is returned if the list is null or all of its elements are null.

Syntax: all(variable IN list WHERE predicate)

Returns:
A Boolean.
### Name

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>An expression that returns a list. A single element cannot be explicitly passed as a literal in the cypher statement. However, an implicit conversion will happen for single elements when passing node properties during cypher execution.</td>
</tr>
<tr>
<td>variable</td>
<td>A variable that can be used from within the predicate.</td>
</tr>
<tr>
<td>predicate</td>
<td>A predicate that is tested against all items in the list.</td>
</tr>
</tbody>
</table>

### Query

```cypher
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 232. Result

<table>
<thead>
<tr>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)-[KNOWS,1]-&gt;(2)-[KNOWS,3]-&gt;(3)</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

#### 10.1.2. any()

The function `any()` returns `true` if the predicate holds for at least one element in the given list. `null` is returned if the list is `null` or all of its elements are `null`.

**Syntax:** `any(variable IN list WHERE predicate)`

**Returns:**

A Boolean.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>An expression that returns a list. A single element cannot be explicitly passed as a literal in the cypher statement. However, an implicit conversion will happen for single elements when passing node properties during cypher execution.</td>
</tr>
<tr>
<td>variable</td>
<td>A variable that can be used from within the predicate.</td>
</tr>
<tr>
<td>predicate</td>
<td>A predicate that is tested against all items in the list.</td>
</tr>
</tbody>
</table>
The query returns nodes with the property `liked_colors` (as a list), where at least one element has the value 'yellow'.

Table 233. Result

<table>
<thead>
<tr>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node[4]{'liked_colors': ['pink', 'yellow', 'black'], 'name': 'Eskil', 'eyes': 'blue', 'age': 41}</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

10.1.3. exists()

The function `exists()` returns `true` if a match for the given pattern exists in the graph, or if the specified property exists in the node, relationship or map. `null` is returned if the input argument is `null`.

Syntax: `exists(pattern-or-property)`

Returns:

A Boolean.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pattern-or-property</td>
<td>A pattern or a property (in the form 'variable.prop').</td>
</tr>
</tbody>
</table>

The names of all nodes with the `name` property are returned, along with a boolean (`true` or `false`) indicating if they are married.

Table 234. Result

<table>
<thead>
<tr>
<th>name</th>
<th>is_married</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Alice&quot;</td>
<td>false</td>
</tr>
<tr>
<td>&quot;Bob&quot;</td>
<td>true</td>
</tr>
<tr>
<td>&quot;Charlie&quot;</td>
<td>false</td>
</tr>
<tr>
<td>&quot;Daniel&quot;</td>
<td>false</td>
</tr>
<tr>
<td>&quot;Eskil&quot;</td>
<td>false</td>
</tr>
</tbody>
</table>
Three nodes are returned: one with a property `name`, one without a property `name`, and one that does not exist (e.g., is `null`). This query exemplifies the behavior of `exists()` when operating on `null` nodes.

Table 235. Result

<table>
<thead>
<tr>
<th>a_name</th>
<th>b_name</th>
<th>b_has_name</th>
<th>c_name</th>
<th>c_has_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Alice&quot;</td>
<td>&lt;null&gt;</td>
<td>false</td>
<td>&lt;null&gt;</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

Rows: 1

10.1.4. `none()`

The function `none()` returns `true` if the predicate does not hold for any element in the given list. `null` is returned if the list is `null` or all of its elements are `null`.

Syntax: `none(variable IN list WHERE predicate)`

Returns:

A Boolean.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>An expression that returns a list. A single element cannot be explicitly passed as a literal in the cypher statement. However, an implicit conversion will happen for single elements when passing node properties during cypher execution.</td>
</tr>
<tr>
<td>variable</td>
<td>A variable that can be used from within the predicate.</td>
</tr>
<tr>
<td>predicate</td>
<td>A predicate that is tested against all items in the list.</td>
</tr>
</tbody>
</table>
Query

```
MATCH p = (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 236. Result

<table>
<thead>
<tr>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)-[KNOWS,1]-&gt;(2)</td>
</tr>
<tr>
<td>(0)-[KNOWS,1]-&gt;(2)-[KNOWS,3]-&gt;(3)</td>
</tr>
<tr>
<td>Rows: 2</td>
</tr>
</tbody>
</table>

10.1.5. `single()`

The function `single()` returns `true` if the predicate holds for exactly one of the elements in the given list. `null` is returned if the list is `null` or all of its elements are `null`.

**Syntax:** `single(variable IN list WHERE predicate)`

**Returns:**

A Boolean.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>An expression that returns a list.</td>
</tr>
<tr>
<td>variable</td>
<td>A variable that can be used from within the predicate.</td>
</tr>
<tr>
<td>predicate</td>
<td>A predicate that is tested against all items in the list.</td>
</tr>
</tbody>
</table>

Query

```
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 237. Result

<table>
<thead>
<tr>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)-[KNOWS,0]-&gt;(1)</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>
10.2. Scalar functions

Scalar functions return a single value.

Functions:

- `coalesce()`
- `endNode()`
- `head()`
- `id()`
- `last()`
- `length()`
- `properties()`
- `randomUUID()`
- `size()`
- `Size of pattern expression`
- `Size of string`
- `startNode()`
- `timestamp()`
- `toBoolean()`
- `toFloat()`
- `toInteger()`
- `type()`

The `length()` and `size()` functions are quite similar, and so it is important to take note of the difference.

Function `length()`
- Only works for paths.

Function `size()`
- Only works for the three types: strings, lists, and pattern expressions.
Graph

10.2.1. coalesce()

The function coalesce() returns the first non-null value in the given list of expressions.

Syntax: coalesce(expression [ , expression]*)

Returns:

The type of the value returned will be that of the first non-null expression.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>An expression that may return null.</td>
</tr>
</tbody>
</table>

Considerations:

null will be returned if all the arguments are null.

Query

```sql
MATCH (a)
WHERE a.name = 'Alice'
RETURN coalesce(a.hairColor, a.eyes)
```

Table 238. Result

<table>
<thead>
<tr>
<th>coalesce(a.hairColor, a.eyes)</th>
<th>&quot;brown&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

10.2.2. endNode()

The function endNode() returns the end node of a relationship.
Syntax: `endNode(relationship)`

Returns:

A Node.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>relationship</td>
<td>An expression that returns a relationship.</td>
</tr>
</tbody>
</table>

Considerations:

`endNode(null) returns null.`

Query

```
MATCH (x:Developer)-[r]-()
RETURN endNode(r)
```

Table 239. Result

<table>
<thead>
<tr>
<th>endNode(r)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Node[2]{name:&quot;Charlie&quot;,eyes:&quot;green&quot;,age:53}</td>
<td></td>
</tr>
<tr>
<td>Node[1]{name:&quot;Bob&quot;,eyes:&quot;blue&quot;,age:25}</td>
<td></td>
</tr>
<tr>
<td>Rows: 2</td>
<td></td>
</tr>
</tbody>
</table>

10.2.3. head()

The function `head()` returns the first element in a list.

Syntax: `head(expression)`

Returns:

The type of the value returned will be that of the first element of the list.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>An expression that returns a list.</td>
</tr>
</tbody>
</table>

Considerations:

`head(null) returns null.`

`head([]) returns null.`

If the first element in `list` is `null`, `head(list)` will return `null`. 
Query

```
MATCH (a)
WHERE a.name = 'Eskil'
RETURN a.liked_colors, head(a.liked_colors)
```

The first element in the list is returned.

Table 240. Result

<table>
<thead>
<tr>
<th>a.liked_colors</th>
<th>head(a.liked_colors)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;pink&quot;, &quot;yellow&quot;, &quot;black&quot;</td>
<td>&quot;pink&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

10.2.4. id()

The function `id()` returns a node or a relationship identifier, unique by an object type and a database. Therefore, it is perfectly allowable for `id()` to return the same value for both nodes and relationships in the same database. For examples on how to get a node and a relationship by ID, see Get node or relationship by id.

Neo4j implements the id so that:

Node

Every node in a database has an identifier. The identifier for a node is guaranteed to be unique among other nodes’ identifiers in the same database, within the scope of a single transaction.

Relationship

Every relationship in a database has an identifier. The identifier for a relationship is guaranteed to be unique among other relationships' identifiers in the same database, within the scope of a single transaction.

Syntax: `id(expression)`

Returns:

An Integer.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>An expression that returns a node or a relationship.</td>
</tr>
</tbody>
</table>

Considerations:

`id(null)` returns null.
Query

```
MATCH (a)
RETURN id(a)
```

The node identifier for each of the nodes is returned.

**Table 241. Result**

<table>
<thead>
<tr>
<th>id(a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

Rows: 5

10.2.5. `last()`

The function `last()` returns the last element in a list.

**Syntax:** `last(expression)`

**Returns:**

The type of the value returned will be that of the last element of the list.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>An expression that returns a list.</td>
</tr>
</tbody>
</table>

**Considerations:**

- `last(null)` returns `null`.
- `last([])` returns `null`.
- If the last element in `list` is `null`, `last(list)` will return `null`.

Query

```
MATCH (a)
WHERE a.name = 'Eskil'
RETURN a.liked_colors, last(a.liked_colors)
```

The last element in the list is returned.

**Table 242. Result**
10.2.6. length()

The function `length()` returns the length of a path.

**Syntax:** `length(path)`

**Returns:**

An Integer.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>An expression that returns a path.</td>
</tr>
</tbody>
</table>

**Considerations:**

`length(null)` returns `null`.

**Query**

```plaintext
MATCH p = (a)-->(b)-->(c)
WHERE a.name = 'Alice'
RETURN length(p)
```

The length of the path `p` is returned.

**Table 243. Result**

<table>
<thead>
<tr>
<th>length(p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

Rows: 3

10.2.7. properties()

The function `properties()` returns a map containing all the properties; the function can be utilized for a relationship or a node. If the argument is already a map, it is returned unchanged.

**Syntax:** `properties(expression)`

**Returns:**
A Map.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>An expression that returns a relationship, a node, or a map.</td>
</tr>
</tbody>
</table>

Considerations:

properties(null) returns null.

Query

```cypher
CREATE (p:Person {name: 'Stefan', city: 'Berlin'})
RETURN properties(p)
```

Table 244. Result

<table>
<thead>
<tr>
<th>properties(p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>{city -&gt; &quot;Berlin&quot;, name -&gt; &quot;Stefan&quot;}</td>
</tr>
</tbody>
</table>

Rows: 1
- Nodes created: 1
- Properties set: 2
- Labels added: 1

10.2.8. randomUUID()

The function `randomUUID()` returns a randomly-generated Universally Unique Identifier (UUID), also known as a Globally Unique Identifier (GUID). This is a 128-bit value with strong guarantees of uniqueness.

Syntax: `randomUUID()`

Returns: A String.

Query

```cypher
RETURN randomUUID() AS uuid
```

Table 245. Result

<table>
<thead>
<tr>
<th>uuid</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;ec83fcd0-ec2e-4be1-98cb-cc0adfb7a1&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

A randomly-generated UUID is returned.
10.2.9. size()

The function size() returns the number of elements in a list.

Syntax: size(list)

Returns:

An Integer.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>An expression that returns a list.</td>
</tr>
</tbody>
</table>

Considerations:

size(null) returns null.

Query

```
RETURN size(['Alice', 'Bob'])
```

Table 246. Result

| size(['Alice', 'Bob']) | 2 | Rows: 1 |

The number of elements in the list is returned.

10.2.10. size() applied to pattern expression

This is the same function size() as described above, but you pass in a pattern expression, instead of a list. The function size will then calculate on a list of paths.

Syntax: size(pattern expression)

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pattern expression</td>
<td>A pattern expression that returns a list.</td>
</tr>
</tbody>
</table>

Query

```
MATCH (a)
WHERE a.name = 'Alice'
RETURN size((a)--()'--() AS fof
```
Table 247. Result

<table>
<thead>
<tr>
<th>fof</th>
<th>3</th>
</tr>
</thead>
</table>

Rows: 1

The number of paths matching the pattern expression is returned. (The size of the list of paths).

10.2.11. size() applied to string

The function \( \text{size}() \) returns the number of Unicode characters in a string.

**Syntax:** \( \text{size}(\text{string}) \)

**Returns:**

An Integer.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>An expression that returns a string value.</td>
</tr>
</tbody>
</table>

**Considerations:**

\( \text{size}(\text{null}) \) returns \text{null}.  

**Query**

\[
\text{MATCH}\ (a) \\
\text{WHERE } \text{size}(a.\text{name}) > 6 \\
\text{RETURN } \text{size}(a.\text{name})
\]

Table 248. Result

<table>
<thead>
<tr>
<th>size(a.name)</th>
<th>7</th>
</tr>
</thead>
</table>

Rows: 1

The number of characters in the string 'Charlie' is returned.

10.2.12. startNode()

The function \( \text{startNode}() \) returns the start node of a relationship.

**Syntax:** \( \text{startNode}(\text{relationship}) \)

**Returns:**
A Node.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>relationship</td>
<td>An expression that returns a relationship.</td>
</tr>
</tbody>
</table>

Considerations:

- `startNode(null)` returns `null`.

Query

```
MATCH (x:Developer)-[r]-()
RETURN startNode(r)
```

Table 249. Result

<table>
<thead>
<tr>
<th>startNode(r)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node[0]{name:&quot;Alice&quot;,eyes:&quot;brown&quot;,age:38}</td>
</tr>
<tr>
<td>Node[0]{name:&quot;Alice&quot;,eyes:&quot;brown&quot;,age:38}</td>
</tr>
</tbody>
</table>

Rows: 2

10.2.13. `timestamp()`

The function `timestamp()` returns the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC.

- It is the equivalent of `datetime().epochMillis`.

Syntax: `timestamp()`

Returns:

- An Integer.

Considerations:

- `timestamp()` will return the same value during one entire query, even for long-running queries.

Query

```
RETURN timestamp()
```

The time in milliseconds is returned.

Table 250. Result
10.2.14. toBoolean()

The function `toBoolean()` converts a string value to a boolean value.

**Syntax:** `toBoolean(expression)`

**Returns:**

A Boolean.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>An expression that returns a boolean or a string value.</td>
</tr>
</tbody>
</table>

**Considerations:**

- `toBoolean(null)` returns `null`.
- If `expression` is a boolean value, it will be returned unchanged.
- If the parsing fails, `null` will be returned.

**Query**

```
RETURN toBoolean('true'), toBoolean('not a boolean')
```

**Table 251. Result**

<table>
<thead>
<tr>
<th>toBoolean('true')</th>
<th>toBoolean('not a boolean')</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

Rows: 1

10.2.15. toFloat()

The function `toFloat()` converts an integer or a string value to a floating point number.

**Syntax:** `toFloat(expression)`

**Returns:**

A Float.

**Arguments:**
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>An expression that returns a numeric or a string value.</td>
</tr>
</tbody>
</table>

**Considerations:**

- `toFloat(null)` returns `null`.
- If `expression` is a floating point number, it will be returned unchanged.
- If the parsing fails, `null` will be returned.

**Query**

```sql
RETURN toFloat('11.5'), toFloat('not a number')
```

**Table 252. Result**

<table>
<thead>
<tr>
<th>toFloat('11.5')</th>
<th>toFloat('not a number')</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.5</td>
<td>&lt;null&gt;</td>
</tr>
</tbody>
</table>

**10.2.16. toInteger()**

The function `toInteger()` converts a floating point or a string value to an integer value.

**Syntax:** `toInteger(expression)`

**Returns:**

An Integer.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>An expression that returns a numeric or a string value.</td>
</tr>
</tbody>
</table>

**Considerations:**

- `toInteger(null)` returns `null`.
- If `expression` is an integer value, it will be returned unchanged.
- If the parsing fails, `null` will be returned.

**Query**

```sql
RETURN toInteger('42'), toInteger('not a number')
```

**Table 253. Result**
10.2.17. type()

The function type() returns the string representation of the relationship type.

Syntax: type(relationship)

Returns:

A String.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>relationship</td>
<td>An expression that returns a relationship.</td>
</tr>
</tbody>
</table>

Considerations:

type(null) returns null.

Query

MATCH (n)-[r]->() WHERE n.name = 'Alice' RETURN type(r)

The relationship type of r is returned.

Table 254. Result

<table>
<thead>
<tr>
<th>type(r)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;KNOWS&quot;</td>
</tr>
<tr>
<td>&quot;KNOWS&quot;</td>
</tr>
</tbody>
</table>

Rows: 2

10.3. Aggregating functions

Aggregating functions take a set of values and calculate an aggregated value over them.

Functions:

- avg() - Numeric values
- avg() - 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 \( \text{count(*)} \). The first, \( n \), is not an aggregate function, so it will be the grouping key. The latter, \( \text{count(*)} \) is an aggregate expression. The matching paths will be divided into different buckets, depending on the grouping key. The aggregate function will then be run on these buckets, calculating an aggregate value per bucket.

To use aggregations to sort the result set, the aggregation must be included in the `RETURN` to be used in the `ORDER BY`.

The `DISTINCT` operator works in conjunction with aggregation. It is used to make all values unique before running them through an aggregate function. More information about `DISTINCT` may be found in Syntax → Aggregation operators.

The following graph is used for the examples below:
10.3.1. avg() - Numeric values

The function \texttt{avg()} returns the average of a set of numeric values.

\textbf{Syntax:} \texttt{avg(expression)}

\textbf{Returns:}

Either an Integer or a Float, depending on the values returned by \texttt{expression} and whether or not the calculation overflows.

\textbf{Arguments:}

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>An expression returning a set of numeric values.</td>
</tr>
</tbody>
</table>

\textbf{Considerations:}

Any \texttt{null} values are excluded from the calculation.

\texttt{avg(null)} returns \texttt{null}.

\textbf{Query}

\begin{verbatim}
MATCH (n:Person)
RETURN avg(n.age)
\end{verbatim}

The average of all the values in the property \texttt{age} is returned.

\textbf{Table 255. Result}

<table>
<thead>
<tr>
<th>avg(n.age)</th>
<th>30.0</th>
</tr>
</thead>
</table>

Rows: 1

10.3.2. avg() - Durations

The function \texttt{avg()} returns the average of a set of Durations.

\textbf{Syntax:} \texttt{avg(expression)}

\textbf{Returns:}

A Duration.

\textbf{Arguments:}

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>An expression returning a set of Durations.</td>
</tr>
</tbody>
</table>
Considerations:

Any null values are excluded from the calculation.

avg(null) returns null.

Query

UNWIND [duration('P2DT3H'), duration('PT1H45S')] AS dur
RETURN avg(dur)

The average of the two supplied Durations is returned.

Table 256. Result

<table>
<thead>
<tr>
<th>avg(dur)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1DT2H22.5S</td>
</tr>
</tbody>
</table>

Rows: 1

10.3.3. collect()

The function collect() returns a single aggregated list containing the values returned by an expression.

Syntax: collect(expression)

Returns:

A list containing heterogeneous elements; the types of the elements are determined by the values returned by expression.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>An expression returning a set of values.</td>
</tr>
</tbody>
</table>

Considerations:

Any null values are ignored and will not be added to the list.

collect(null) returns an empty list.

Query

MATCH (n:Person)
RETURN collect(n.age)

All the values are collected and returned in a single list.

Table 257. Result
10.3.4. count()

The function `count()` returns the number of values or rows, and appears in two variants:

- `count(*)`
  - returns the number of matching rows.

- `count(expr)`
  - returns the number of non-null values returned by an expression.

**Syntax:** `count(expression)`

**Returns:**

An Integer.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>An expression.</td>
</tr>
</tbody>
</table>

**Considerations:**

- `count(*)` includes rows returning null.
- `count(expr)` ignores null values.
- `count(null)` returns 0.

**Using count(*) to return the number of nodes**

The function `count(*)` can be used to return the number of nodes; for example, the number of nodes connected to some node `n`.

**Query**

```
MATCH (n {name: 'A'})-->(x)
RETURN labels(n), n.age, count(*)
```

The labels and age property of the start node `n` and the number of nodes related to `n` are returned.

**Table 258. Result**

<table>
<thead>
<tr>
<th>labels(n)</th>
<th>n.age</th>
<th>count(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[&quot;Person&quot;]</td>
<td>13</td>
<td>4</td>
</tr>
</tbody>
</table>
Using `count(*)` to group and count relationship types

The function `count(*)` can be used to group the type of matched relationships and return the number.

Query

```
MATCH (n {name: 'A'})-[r]->()
RETURN type(r), count(*)
```

The type of matched relationships are grouped and the group count are returned.

Table 259. Result

<table>
<thead>
<tr>
<th>type(r)</th>
<th>count(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;KNOWS&quot;</td>
<td>3</td>
</tr>
<tr>
<td>&quot;READS&quot;</td>
<td>1</td>
</tr>
</tbody>
</table>

Rows: 2

Using `count(expression)` to return the number of values

Instead of simply returning the number of rows with `count(*)`, it may be more useful to return the actual number of values returned by an expression.

Query

```
MATCH (n {name: 'A'})-->()
RETURN count(x)
```

The number of nodes that are connected directly (one relationship) to the node, with the name 'A', is returned.

Table 260. Result

<table>
<thead>
<tr>
<th>count(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

Rows: 1

Counting non-null values

The function `count(expression)` can be used to return the number of non-null values returned by the expression.

Query

```
MATCH (n:Person)
RETURN count(n.age)
```

216
The number of nodes with the label Person and a property age is returned. (If you want the sum, use sum(n.age))

Table 261. Result

<table>
<thead>
<tr>
<th>count(n.age)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

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.

Query

```sql
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 262. Result

<table>
<thead>
<tr>
<th>count(DISTINCT friend_of_friend)</th>
<th>count(friend_of_friend)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

10.3.5. max()

The function max() returns the maximum value in a set of values.

Syntax: max(expression)

Returns:

A property type, or a list, depending on the values returned by expression.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>An expression returning a set containing any combination of property types and lists thereof.</td>
</tr>
</tbody>
</table>
Considerations:

Any null values are excluded from the calculation.

In a mixed set, any numeric value is always considered to be higher than any string value, and any string value is always considered to be higher than any list.

Lists are compared in dictionary order, i.e. list elements are compared pairwise in ascending order from the start of the list to the end.

max(null) returns null.

Query

UNWIND [1, 'a', null, 0.2, 'b', '1', '99'] AS val
RETURN max(val)

The highest of all the values in the mixed set — in this case, the numeric value 1 — is returned.

Table 263. Result

<table>
<thead>
<tr>
<th>max(val)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Rows: 1

Query

UNWIND [[1, 'a', 89], [1, 2]] AS val
RETURN max(val)

The highest of all the lists in the set — in this case, the list [1, 2] — is returned, as the number 2 is considered to be a higher value than the string 'a', even though the list [1, 'a', 89] contains more elements.

Table 264. Result

<table>
<thead>
<tr>
<th>max(val)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,2]</td>
</tr>
</tbody>
</table>

Rows: 1

Query

MATCH (n:Person)
RETURN max(n.age)

The highest of all the values in the property age is returned.

Table 265. Result
10.3.6. \texttt{min()}

The function \texttt{min()} returns the minimum value in a set of values.

\textbf{Syntax:} \texttt{min(expression)}

\textbf{Returns:}

A property type, or a list, depending on the values returned by \texttt{expression}.

\textbf{Arguments:}

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{expression}</td>
<td>An expression returning a set containing any combination of property types and lists thereof.</td>
</tr>
</tbody>
</table>

\textbf{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.
- \texttt{min(null)} returns \texttt{null}.

\textbf{Query}

```sql
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.

\textbf{Table 266. Result}

<table>
<thead>
<tr>
<th>min(val)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;1&quot;</td>
</tr>
</tbody>
</table>

Rows: 1
Query

\[
\text{UNWIND } ['d', [1, 2], ['a', 'c', 23]] \text{ AS } \text{val}
\]
\[
\text{RETURN } \text{min}(\text{val})
\]

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 267. Result

<table>
<thead>
<tr>
<th>min(val)</th>
</tr>
</thead>
<tbody>
<tr>
<td>['a', 'c', 23]</td>
</tr>
</tbody>
</table>

Rows: 1

Query

\[
\text{MATCH } (\text{n:Person})
\]
\[
\text{RETURN } \text{min}(\text{n.age})
\]

The lowest of all the values in the property age is returned.

Table 268. Result

<table>
<thead>
<tr>
<th>min(n.age)</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
</tr>
</tbody>
</table>

Rows: 1

10.3.7. percentileCont()

The function percentileCont() returns the percentile of the given value over a group, with a percentile from 0.0 to 1.0. It uses a linear interpolation method, calculating a weighted average between two values if the desired percentile lies between them. For nearest values using a rounding method, see percentileDisc.

Syntax: percentileCont(expression, percentile)

Returns:

A Float.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression.</td>
</tr>
<tr>
<td>percentile</td>
<td>A numeric value between 0.0 and 1.0</td>
</tr>
</tbody>
</table>

Considerations:

220
Any null values are excluded from the calculation.

percentileCont(null, percentile) returns null.

Query

MATCH (n:Person)
RETURN percentileCont(n.age, 0.4)

The 40th percentile of the values in the property age is returned, calculated with a weighted average.

Table 269. Result

<table>
<thead>
<tr>
<th>percentileCont(n.age, 0.4)</th>
<th>29.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

10.3.8. percentileDisc()

The function percentileDisc() returns the percentile of the given value over a group, with a percentile from 0.0 to 1.0. It uses a rounding method and calculates the nearest value to the percentile. For interpolated values, see percentileCont.

Syntax: percentileDisc(expression, percentile)

Returns:

Either an Integer or a Float, depending on the values returned by expression and whether or not the calculation overflows.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression.</td>
</tr>
<tr>
<td>percentile</td>
<td>A numeric value between 0.0 and 1.0</td>
</tr>
</tbody>
</table>

Considerations:

Any null values are excluded from the calculation.

percentileDisc(null, percentile) returns null.

Query

MATCH (n:Person)
RETURN percentileDisc(n.age, 0.5)

The 50th percentile of the values in the property age is returned.

Table 270. Result
10.3.9. stDev()

The function `stDev()` returns the standard deviation for the given value over a group. It uses a standard two-pass method, with $N - 1$ as the denominator, and should be used when taking a sample of the population for an unbiased estimate. When the standard variation of the entire population is being calculated, `stDevP` should be used.

Syntax: `stDev(expression)`

Returns:

A Float.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression.</td>
</tr>
</tbody>
</table>

Considerations:

Any null values are excluded from the calculation.

`stDev(null) returns 0`.

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 271. Result

<table>
<thead>
<tr>
<th>stDev(n.age)</th>
<th>15.716233645501712</th>
</tr>
</thead>
</table>

Rows: 1

10.3.10. stDevP()

The function `stDevP()` returns the standard deviation for the given value over a group. It uses a standard two-pass method, with $N$ as the denominator, and should be used when calculating the standard deviation for an entire population. When the standard variation of only a sample of the population is being calculated, `stDev` should be used.
Syntax: \texttt{stDevP(expression)}

Returns:
A Float.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{expression}</td>
<td>A numeric expression.</td>
</tr>
</tbody>
</table>

Considerations:

Any null values are excluded from the calculation.

\texttt{stDevP(null)} returns 0.

Query

\begin{verbatim}
MATCH (n)
WHERE n.name IN ['A', 'B', 'C']
RETURN stDevP(n.age)
\end{verbatim}

The population standard deviation of the values in the property \texttt{age} is returned.

Table 272. Result

<table>
<thead>
<tr>
<th>\texttt{stDevP(n.age)}</th>
<th>12.832251036613439</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

10.3.11. \texttt{sum()} - Numeric values

The function \texttt{sum()} returns the sum of a set of numeric values.

Syntax: \texttt{sum(expression)}

Returns:

Either an Integer or a Float, depending on the values returned by \texttt{expression}.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{expression}</td>
<td>An expression returning a set of numeric values.</td>
</tr>
</tbody>
</table>

Considerations:

Any null values are excluded from the calculation.
sum(null) returns 0.

Query

```sql
MATCH (n:Person)
RETURN sum(n.age)
```

The sum of all the values in the property `age` is returned.

Table 273. Result

<table>
<thead>
<tr>
<th>sum(n.age)</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

10.3.12. sum() - Durations

The function `sum()` returns the sum of a set of durations.

Syntax: `sum(expression)`

Returns:

- A Duration.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>An expression returning a set of Durations.</td>
</tr>
</tbody>
</table>

Considerations:

Any null values are excluded from the calculation.

Query

```sql
UNWIND [duration('P2DT3H'), duration('PT1H45S')] AS dur
RETURN sum(dur)
```

The sum of the two supplied Durations is returned.

Table 274. Result

<table>
<thead>
<tr>
<th>sum(dur)</th>
<th>P2DT4H45S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>
10.4. List functions

List functions return lists of things — nodes in a path, and so on.

Further details and examples of lists may be found in Lists and List operators.

Functions:

- keys()
- labels()
- nodes()
- range()
- reduce()
- relationships()
- reverse()
- tail()

Graph

10.4.1. keys()

keys returns a list containing the string representations for all the property names of a node, relationship, or map.

Syntax: keys(expression)

Returns:

A list containing String elements.

Arguments:
### Considerations:

| expression | An expression that returns a node, a relationship, or a map. |

| keys(null) returns null. |

### Query

```sql
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 275. Result

<table>
<thead>
<tr>
<th>keys(a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[&quot;name&quot;,&quot;eyes&quot;,&quot;age&quot;]</td>
</tr>
</tbody>
</table>

Rows: 1

### 10.4.2. labels()

**labels** returns a list containing the string representations for all the labels of a node.

**Syntax:** `labels(node)`

**Returns:**

A list containing String elements.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>An expression that returns a single node.</td>
</tr>
</tbody>
</table>

**Considerations:**

| labels(null) returns null. |

**Query**

```sql
MATCH (a) WHERE a.name = 'Alice'
RETURN labels(a)
```

A list containing all the labels of the node bound to `a` is returned.

#### Table 276. Result
10.4.3. nodes()

`nodes()` returns a list containing all the nodes in a path.

**Syntax:** `nodes(path)`

**Returns:**

A list containing Node elements.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>An expression that returns a path.</td>
</tr>
</tbody>
</table>

**Considerations:**

`nodes(null)` returns `null`.

**Query**

```sql
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 277. Result

<table>
<thead>
<tr>
<th>nodes(p)</th>
</tr>
</thead>
</table>

Rows: 1

10.4.4. range()

`range()` returns a list comprising all integer values within a range bounded by a start value `start` and end value `end`, where the difference `step` between any two consecutive values is constant; i.e. an arithmetic progression. To create ranges with decreasing integer values, use a negative value `step`. The range is inclusive for non-empty ranges, and the arithmetic progression will therefore always contain `start` and — depending on the values of `start`, `step` and `end` — `end`. The only exception where the range does not contain `start` are empty ranges. An empty range will be returned if the value `step` is negative and `start - end` is positive, or vice versa, e.g. `range(0, 5, -1)`.
Syntax: \texttt{range}(start, end [, step])

Returns:

A list of Integer elements.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>An expression that returns an integer value.</td>
</tr>
<tr>
<td>end</td>
<td>An expression that returns an integer value.</td>
</tr>
<tr>
<td>step</td>
<td>A numeric expression defining the difference between any two consecutive values, with a default of $1$.</td>
</tr>
</tbody>
</table>

Query

\begin{verbatim}
RETURN range(0, 10), range(2, 18, 3), range(0, 5, -1)
\end{verbatim}

Three lists of numbers in the given ranges are returned.

Table 278. Result

<table>
<thead>
<tr>
<th>range(0, 10)</th>
<th>range(2, 18, 3)</th>
<th>range(0, 5, -1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0,1,2,3,4,5,6,7,8,9,10]</td>
<td>[2,5,8,11,14,17]</td>
<td>[]</td>
</tr>
</tbody>
</table>

Rows: 1

10.4.5. \texttt{reduce()}

\texttt{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 \texttt{fold} or \texttt{reduce} method in functional languages such as Lisp and Scala.

Syntax: \texttt{reduce}(accumulator = initial, variable IN list | expression)

Returns:

The type of the value returned depends on the arguments provided, along with the semantics of \texttt{expression}.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>accumulator</td>
<td>A variable that will hold the result and the partial results as the list is iterated.</td>
</tr>
<tr>
<td>initial</td>
<td>An expression that runs once to give a starting value to the accumulator.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>list</td>
<td>An expression that returns a list.</td>
</tr>
<tr>
<td>variable</td>
<td>The closure will have a variable introduced in its context. We decide here which variable to use.</td>
</tr>
<tr>
<td>expression</td>
<td>This expression will run once per value in the list, and produce the result value.</td>
</tr>
</tbody>
</table>

### Query

```
MATCH p = (a)-->(b)-->(c)
WHERE a.name = 'Alice' AND b.name = 'Bob' AND c.name = 'Daniel'
RETURN reduce(totalAge = 0, n IN nodes(p) | totalAge + n.age) AS reduction
```

The `age` property of all nodes in the path are summed and returned as a single value.

#### Table 279. Result

<table>
<thead>
<tr>
<th>reduction</th>
<th>117</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows:</td>
<td>1</td>
</tr>
</tbody>
</table>

#### 10.4.6. relationships()

`relationships()` returns a list containing all the relationships in a path.

**Syntax:** `relationships(path)`

**Returns:**

A list containing Relationship elements.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>An expression that returns a path.</td>
</tr>
</tbody>
</table>

**Considerations:**

`relationships(null)` returns `null`.

#### 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 280. Result
10.4.7. reverse()

reverse() returns a list in which the order of all elements in the original list have been reversed.

**Syntax:** reverse(original)

**Returns:**

A list containing homogeneous or heterogeneous elements; the types of the elements are determined by the elements within original.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>An expression that returns a list.</td>
</tr>
</tbody>
</table>

**Considerations:**

Any null element in original is preserved.

**Query**

```
WITH [4923, 'abc', 521, null, 487] AS ids
RETURN reverse(ids)
```

**Table 281. Result**

<table>
<thead>
<tr>
<th>reverse(ids)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[487,&lt;null&gt;,521,&quot;abc&quot;,4923]</td>
<td></td>
</tr>
</tbody>
</table>

Rows: 1

10.4.8. tail()

tail() returns a list 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:**
### list

An expression that returns a list.

#### Query

```
MATCH (a) WHERE a.name = 'Eskil'
RETURN a.array, tail(a.array)
```

The property named array and a list comprising all but the first element of the array property are returned.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a.array</td>
<td>tail(a.array)</td>
</tr>
<tr>
<td>&quot;one&quot;,&quot;two&quot;,&quot;three&quot;</td>
<td>&quot;two&quot;,&quot;three&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

### 10.5. Mathematical functions - numeric

These functions all operate on numeric expressions only, and will return an error if used on any other values. See also Mathematical operators.

#### Functions:

- abs()
- ceil()
- floor()
- rand()
- round()
- sign()

The following graph is used for the examples below:
Graph

10.5.1. abs()

abs() returns the absolute value of the given number.

Syntax: abs(expression)

Returns:

The type of the value returned will be that of expression.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression.</td>
</tr>
</tbody>
</table>

Considerations:

- abs(null) returns null.
- If expression is negative, -(expression) (i.e. the negation of expression) is returned.

Query

```
MATCH (a), (e) WHERE a.name = 'Alice' AND e.name = 'Eskil' RETURN a.age, e.age, abs(a.age - e.age)
```

The absolute value of the age difference is returned.

Table 283. Result

<table>
<thead>
<tr>
<th>a.age</th>
<th>e.age</th>
<th>abs(a.age - e.age)</th>
</tr>
</thead>
<tbody>
<tr>
<td>38</td>
<td>41</td>
<td>3</td>
</tr>
</tbody>
</table>

Rows: 1
10.5.2. ceil()

`ceil()` returns the smallest floating point number that is greater than or equal to the given number and equal to a mathematical integer.

**Syntax:** `ceil(expression)`

**Returns:**

A Float.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression.</td>
</tr>
</tbody>
</table>

**Considerations:**

`ceil(null)` returns null.

**Query**

```
RETURN ceil(0.1)
```

The ceil of 0.1 is returned.

Table 284. Result

<table>
<thead>
<tr>
<th>ceil(0.1)</th>
<th>1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows:</td>
<td>1</td>
</tr>
</tbody>
</table>

10.5.3. floor()

`floor()` returns the largest floating point number that is less than or equal to the given number and equal to a mathematical integer.

**Syntax:** `floor(expression)`

**Returns:**

A Float.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression.</td>
</tr>
</tbody>
</table>
Considerations:

floor(null) returns null.

Query

```
RETURN floor(0.9)
```

The floor of 0.9 is returned.

Table 285. Result

<table>
<thead>
<tr>
<th>floor(0.9)</th>
<th>0.0</th>
</tr>
</thead>
</table>

Rows: 1

10.5.4. rand()

rand() returns a random floating point number in the range from 0 (inclusive) to 1 (exclusive); i.e. [0, 1). The numbers returned follow an approximate uniform distribution.

Syntax: rand()

Returns:

A Float.

Query

```
RETURN rand()
```

A random number is returned.

Table 286. Result

<table>
<thead>
<tr>
<th>rand()</th>
<th>0.5066942053904978</th>
</tr>
</thead>
</table>

Rows: 1

10.5.5. round()

round() returns the value of the given number rounded to the nearest integer.

Syntax: round(expression)

Returns:

A Float.
Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression.</td>
</tr>
</tbody>
</table>

Considerations:

round(null) returns null.

Query

```sql
RETURN round(3.141592)
```

3.0 is returned.

Table 287. Result

<table>
<thead>
<tr>
<th>round(3.141592)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.0</td>
</tr>
</tbody>
</table>

10.5.6. sign()

sign() returns the signum of the given number: 0 if the number is 0, -1 for any negative number, and 1 for any positive number.

Syntax: sign(expression)

Returns:

An Integer.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression.</td>
</tr>
</tbody>
</table>

Considerations:

sign(null) returns null.

Query

```sql
RETURN sign(-17), sign(0.1)
```

The signs of -17 and 0.1 are returned.

Table 288. Result
10.6. Mathematical functions - logarithmic

These functions all operate on numeric expressions only, and will return an error if used on any other values. See also Mathematical operators.

Functions:

- `e()`
- `exp()`
- `log()`
- `log10()`
- `sqrt()`

10.6.1. e()

e() returns the base of the natural logarithm, e.

Syntax: `e()`

Returns:

A Float.

Query

```plaintext
RETURN e()
```

The base of the natural logarithm, e, is returned.

Table 289. Result

<table>
<thead>
<tr>
<th>e()</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2.718281828459045</td>
</tr>
</tbody>
</table>

Rows: 1

10.6.2. exp()

exp() returns e^n, where e is the base of the natural logarithm, and n is the value of the argument expression.
Syntax: \texttt{e(expression)}

Returns: A Float.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression.</td>
</tr>
</tbody>
</table>

Considerations:

\texttt{exp(null)} returns \texttt{null}.

Query

\texttt{RETURN exp(2)}

e to the power of 2 is returned.

Table 290. Result

\begin{tabular}{|c|}
\hline
\texttt{exp(2)}
\\hline
7.38905600993065
\hline
\end{tabular}

Rows: 1

10.6.3. \texttt{log()}

\texttt{log()} returns the natural logarithm of a number.

Syntax: \texttt{log(expression)}

Returns: A Float.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression.</td>
</tr>
</tbody>
</table>

Considerations:

\texttt{log(null)} returns \texttt{null}.

\texttt{log(0)} returns \texttt{null}.
The natural logarithm of 27 is returned.

Table 291. Result

<table>
<thead>
<tr>
<th>log(27)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.29583686604329</td>
</tr>
</tbody>
</table>

Rows: 1

10.6.4. log10()

log10() returns the common logarithm (base 10) of a number.

Syntax: `log10(expression)`

Returns:

A Float.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression.</td>
</tr>
</tbody>
</table>

Considerations:

- log10(null) returns null.
- log10(0) returns null.

The common logarithm of 27 is returned.

Table 292. Result

<table>
<thead>
<tr>
<th>log10(27)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.4313637641589874</td>
</tr>
</tbody>
</table>

Rows: 1

10.6.5. sqrt()

sqrt() returns the square root of a number.
Syntax: \( \text{sqrt(expression)} \)

Returns:

A Float.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression.</td>
</tr>
</tbody>
</table>

Considerations:

- \( \text{sqrt(null)} \) returns null.
- \( \text{sqrt(<any negative number>)} \) returns null

Query

```
RETURN sqrt(256)
```

The square root of 256 is returned.

Table 293. Result

<table>
<thead>
<tr>
<th>sqrt(256)</th>
<th>16.0</th>
</tr>
</thead>
</table>

Rows: 1

10.7. Mathematical functions - trigonometric

These functions all operate on numeric expressions only, and will return an error if used on any other values. See also Mathematical operators.

Functions:

- acos()
- asin()
- atan()
- atan2()
- cos()
- cot()
- degrees()
- haversin()
• Spherical distance using the `haversin()` function

• `pi()`

• `radians()`

• `sin()`

• `tan()`

### 10.7.1. `acos()`

`acos()` returns the arccosine of a number in radians.

**Syntax:** `acos(expression)`

**Returns:**

A Float.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression that represents the angle in radians.</td>
</tr>
</tbody>
</table>

**Considerations:**

- `acos(null)` returns `null`.
- If `(expression < -1)` or `(expression > 1)`, then `(acos(expression))` returns `null`.

**Query**

```sql
RETURN acos(0.5)
```

The arccosine of 0.5 is returned.

**Table 294. Result**

<table>
<thead>
<tr>
<th>acos(0.5)</th>
<th>1.0471975511965979</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

### 10.7.2. `asin()`

`asin()` returns the arcsine of a number in radians.

**Syntax:** `asin(expression)`

**Returns:**
A Float.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression that represents the angle in radians.</td>
</tr>
</tbody>
</table>

Considerations:

asin(null) returns null.

If (expression < -1) or (expression > 1), then (asin(expression)) returns null.

Query

```
RETURN asin(0.5)
```

The arcsine of 0.5 is returned.

Table 295. Result

<table>
<thead>
<tr>
<th>asin(0.5)</th>
<th>0.5235987755982989</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

10.7.3. atan()

atan() returns the arctangent of a number in radians.

Syntax: atan(expression)

Returns:

A Float.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression that represents the angle in radians.</td>
</tr>
</tbody>
</table>

Considerations:

atan(null) returns null.

Query

```
RETURN atan(0.5)
```
The arctangent of 0.5 is returned.

Table 296. Result

<table>
<thead>
<tr>
<th>atan(0.5)</th>
<th>0.4636476090008061</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

10.7.4. atan2()

atan2() returns the arctangent2 of a set of coordinates in radians.

Syntax: atan2(expression1, expression2)

Returns:

A Float.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression1</td>
<td>A numeric expression for y that represents the angle in radians.</td>
</tr>
<tr>
<td>expression2</td>
<td>A numeric expression for x that represents the angle in radians.</td>
</tr>
</tbody>
</table>

Considerations:

atan2(null, null), atan2(null, expression2) and atan(expression1, null) all return null.

Query

```
RETURN atan2(0.5, 0.6)
```

The arctangent2 of 0.5 and 0.6 is returned.

Table 297. Result

<table>
<thead>
<tr>
<th>atan2(0.5, 0.6)</th>
<th>0.6947382761967033</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

10.7.5. cos()

cos() returns the cosine of a number.

Syntax: cos(expression)
Returns:

A Float.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression that represents the angle in radians.</td>
</tr>
</tbody>
</table>

Considerations:

\[ \cos(\text{null}) \text{ returns null.} \]

Query

\[
\text{RETURN } \cos(0.5)
\]

The cosine of 0.5 is returned.

Table 298. Result

<table>
<thead>
<tr>
<th>cos(0.5)</th>
<th>0.8775825618903728</th>
</tr>
</thead>
</table>

Rows: 1

10.7.6. \( \cot() \)

\( \cot() \) returns the cotangent of a number.

Syntax: \( \cot(\text{expression}) \)

Returns:

A Float.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression that represents the angle in radians.</td>
</tr>
</tbody>
</table>

Considerations:

\[ \cot(\text{null}) \text{ returns null.} \]

\[ \cot(0) \text{ returns null.} \]
The cotangent of 0.5 is returned.

Table 299. Result

<table>
<thead>
<tr>
<th>cot(0.5)</th>
<th>1.830487721712452</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

10.7.7. degrees()

degrees() converts radians to degrees.

Syntax: degrees(expression)

Returns:

A Float.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression that represents the angle in radians.</td>
</tr>
</tbody>
</table>

Considerations:

degrees(null) returns null.

Query

RETURN degrees(3.14159)

The number of degrees in something close to pi is returned.

Table 300. Result

<table>
<thead>
<tr>
<th>degrees(3.14159)</th>
<th>179.9998479605043</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

10.7.8. haversin()

haversin() returns half the versine of a number.

Syntax: haversin(expression)
Returns:

A Float.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression that represents the angle in radians.</td>
</tr>
</tbody>
</table>

Considerations:

haversin(null) returns null.

Query

```
RETURN haversin(0.5)
```

The haversine of 0.5 is returned.

Table 301. Result

<table>
<thead>
<tr>
<th>haversin(0.5)</th>
<th>0.06120871905481362</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows:</td>
<td>1</td>
</tr>
</tbody>
</table>

10.7.9. Spherical distance using the haversin() function

The haversin() function may be used to compute the distance on the surface of a sphere between two points (each given by their latitude and longitude). In this example the spherical distance (in km) between Berlin in Germany (at lat 52.5, lon 13.4) and San Mateo in California (at lat 37.5, lon -122.3) is calculated using an average earth radius of 6371 km.

Query

```
CREATE (ber:City {lat: 52.5, lon: 13.4}), (sm:City {lat: 37.5, lon: -122.3})
RETURN 2 * 6371 * asin(sqrt(haversin(radians(sm.lat - ber.lat))
  + cos(radians(sm.lat)) * cos(radians(ber.lat)) *
  haversin(radians(sm.lon - ber.lon)))) AS dist
```

The estimated distance between 'Berlin' and 'San Mateo' is returned.

Table 302. Result

<table>
<thead>
<tr>
<th>dist</th>
<th>9129.969740051658</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows:</td>
<td>1</td>
</tr>
<tr>
<td>Nodes created:</td>
<td>2</td>
</tr>
<tr>
<td>Properties set:</td>
<td>4</td>
</tr>
<tr>
<td>Labels added:</td>
<td>2</td>
</tr>
</tbody>
</table>
10.7.10. **pi()**

**pi()** returns the mathematical constant \( \pi \).

**Syntax:** `pi()`

**Returns:**

A Float.

**Query**

```
RETURN pi()
```

The constant \( \pi \) is returned.

**Table 303. Result**

<table>
<thead>
<tr>
<th>pi()</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.141592653589793</td>
</tr>
</tbody>
</table>

Rows: 1

10.7.11. **radians()**

**radians()** converts degrees to radians.

**Syntax:** `radians(expression)`

**Returns:**

A Float.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression that represents the angle in degrees.</td>
</tr>
</tbody>
</table>

**Considerations:**

`radians(null)` returns `null`.

**Query**

```
RETURN radians(180)
```

The number of radians in 180 degrees is returned (\( \pi \)).

**Table 304. Result**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>246</td>
<td></td>
</tr>
</tbody>
</table>
10.7.12. sin()

`sin()` returns the sine of a number.

**Syntax:** `sin(expression)`

**Returns:**

A Float.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression that represents the angle in radians.</td>
</tr>
</tbody>
</table>

**Considerations:**

`sin(null)` returns `null`.

**Query**

```
RETURN sin(0.5)
```

The sine of `0.5` is returned.

**Table 305. Result**

<table>
<thead>
<tr>
<th>sin(0.5)</th>
<th>0.479425538604203</th>
</tr>
</thead>
</table>

Rows: 1

10.7.13. tan()

`tan()` returns the tangent of a number.

**Syntax:** `tan(expression)`

**Returns:**

A Float.

**Arguments:**
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>A numeric expression that represents the angle in radians.</td>
</tr>
</tbody>
</table>

Considerations:

\[ \tan(\text{null}) \text{ returns null.} \]

Query

```
RETURN \tan(0.5)
```

The tangent of 0.5 is returned.

Table 306. Result

<table>
<thead>
<tr>
<th>(\tan(0.5))</th>
<th>0.5463024898437905</th>
</tr>
</thead>
</table>

Rows: 1

10.8. String functions

*These functions all operate on string expressions only, and will return an error if used on any other values. The exception to this rule is \(\text{toString()}\), 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 \(\text{toString()}\) 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()
• toUpper()
• trim()

10.8.1. left()

left() returns a string containing the specified number of leftmost characters of the original string.

Syntax: `left(original, length)`

Returns:

A String.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>An expression that returns a string.</td>
</tr>
<tr>
<td>n</td>
<td>An expression that returns a positive integer.</td>
</tr>
</tbody>
</table>

Considerations:

- `left(null, length)` and `left(null, null)` both return `null`
- `left(original, null)` will raise an error.
- If `length` is not a positive integer, an error is raised.
- If `length` exceeds the size of `original`, `original` is returned.

Query

```
RETURN left('hello', 3)
```

Table 307. Result

<table>
<thead>
<tr>
<th><code>left('hello', 3)</code></th>
<th>&quot;hel&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

10.8.2. ltrim()

ltrim() returns the original string with leading whitespace removed.
Syntax: \texttt{lTrim(original)}

Returns:

A String.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>An expression that returns a string.</td>
</tr>
</tbody>
</table>

Considerations:

\texttt{lTrim(null)} returns null

Query

\begin{verbatim}
RETURN lTrim('   hello')
\end{verbatim}

Table 308. Result

<table>
<thead>
<tr>
<th>lTrim(' hello')</th>
<th>&quot;hello&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

10.8.3. replace()

\texttt{replace()} returns a string in which all occurrences of a specified string in the original string have been replaced by another (specified) string.

Syntax: \texttt{replace(original, search, replace)}

Returns:

A String.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>An expression that returns a string.</td>
</tr>
<tr>
<td>search</td>
<td>An expression that specifies the string to be replaced in \texttt{original}.</td>
</tr>
<tr>
<td>replace</td>
<td>An expression that specifies the replacement string.</td>
</tr>
</tbody>
</table>

Considerations:

If any argument is \texttt{null}, \texttt{null} will be returned.
If search is not found in original, original will be returned.

Query

```
RETURN replace("hello", "l", "w")
```

Table 309. Result

<table>
<thead>
<tr>
<th>replace(&quot;hello&quot;, &quot;l&quot;, &quot;w&quot;)</th>
<th>&quot;hewwo&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

10.8.4. reverse()

`reverse()` returns a string in which the order of all characters in the original string have been reversed.

**Syntax:** `reverse(original)`

**Returns:**

A String.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>An expression that returns a string.</td>
</tr>
</tbody>
</table>

**Considerations:**

`reverse(null)` returns null.

Query

```
RETURN reverse('anagram')
```

Table 310. Result

<table>
<thead>
<tr>
<th>reverse('anagram')</th>
<th>&quot;margana&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

10.8.5. right()

`right()` returns a string containing the specified number of rightmost characters of the original string.

**Syntax:** `right(original, length)`

**Returns:**
A String.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>An expression that returns a string.</td>
</tr>
<tr>
<td>n</td>
<td>An expression that returns a positive integer.</td>
</tr>
</tbody>
</table>

Considerations:

- `right(null, length)` and `right(null, null)` both return `null`.
- `right(original, null)` will raise an error.
- If `length` is not a positive integer, an error is raised.
- If `length` exceeds the size of `original`, `original` is returned.

Query

```
RETURN right('hello', 3)
```

Table 311. Result

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>right('hello', 3)</code></td>
<td></td>
</tr>
<tr>
<td>&quot;llo&quot;</td>
<td></td>
</tr>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

10.8.6. rtrim()

`rtrim()` returns the original string with trailing whitespace removed.

Syntax: `rtrim(original)`

Returns:

A String.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>An expression that returns a string.</td>
</tr>
</tbody>
</table>

Considerations:

- `rtrim(null)` returns `null`
10.8.7. split()

**split()** returns a list of strings resulting from the splitting of the original string around matches of the given delimiter.

**Syntax:** `split(original, splitDelimiter)`

**Returns:**
A list of Strings.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>An expression that returns a string.</td>
</tr>
<tr>
<td>splitDelimiter</td>
<td>The string with which to split original.</td>
</tr>
</tbody>
</table>

**Considerations:**

- `split(null, splitDelimiter)` and `split(original, null)` both return null

Query

```
RETURN split('one,two', ',')
```

Table 313. Result

```
split('one,two', ',')
```

["one", "two"]

Rows: 1

10.8.8. substring()

**substring()** returns a substring of the original string, beginning with a 0-based index start and length.

**Syntax:** `substring(original, start [, length])`
Returns:

A String.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>An expression that returns a string.</td>
</tr>
<tr>
<td>start</td>
<td>An expression that returns a positive integer, denoting the position at which the substring will begin.</td>
</tr>
<tr>
<td>length</td>
<td>An expression that returns a positive integer, denoting how many characters of original will be returned.</td>
</tr>
</tbody>
</table>

Considerations:

- start uses a zero-based index.

If length is omitted, the function returns the substring starting at the position given by start and extending to the end of original.

If original is null, null is returned.

If either start or length is null or a negative integer, an error is raised.

If start is 0, the substring will start at the beginning of original.

If length is 0, the empty string will be returned.

Query

```
RETURN substring('hello', 1, 3), substring('hello', 2)
```

Table 314. Result

<table>
<thead>
<tr>
<th>substring('hello', 1, 3)</th>
<th>substring('hello', 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;ell&quot;</td>
<td>&quot;llo&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

10.8.9. toLower()

toLower() returns the original string in lowercase.

Syntax: toLower(original)

Returns:

A String.

Arguments:
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>An expression that returns a string.</td>
</tr>
</tbody>
</table>

**Considerations:**

`toLowerCase(null)` returns `null`

**Query**

```
RETURN toLower('HELLO')
```

**Table 315. Result**

<table>
<thead>
<tr>
<th>toLowerCase('HELLO')</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;hello&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

**10.8.10. toString()**

`toString()` converts an integer, float or boolean value to a string.

**Syntax:** `toString(expression)`

**Returns:**

A String.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>An expression that returns a number, a boolean, or a string.</td>
</tr>
</tbody>
</table>

**Considerations:**

`toString(null)` returns `null`

If `expression` is a string, it will be returned unchanged.

**Query**

```
RETURN toString(11.5),
toString('already a string'),
toString(true),
toString(date({year:1984, month:10, day:11})) AS dateString,
toString(datetime({year:1984, month:10, day:11, hour:12, minute:31, second:14, millisecond: 341, timezone: 'Europe/Stockholm'})) AS datetimeString,
toString(duration({minutes:12, seconds: -60})) AS durationString
```

**Table 316. Result**
10.8.11. toUpper()

toUpper() returns the original string in uppercase.

**Syntax:** toUpper(original)

**Returns:**
A String.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>An expression that returns a string.</td>
</tr>
</tbody>
</table>

**Considerations:**

toUpper(null) returns null

**Query**

```
RETURN toUpper('hello')
```

**Table 317. Result**

<table>
<thead>
<tr>
<th>toUpper('hello')</th>
<th>&quot;HELLO&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

10.8.12. trim()

trim() returns the original string with leading and trailing whitespace removed.

**Syntax:** trim(original)

**Returns:**
A String.

**Arguments:**

256
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>An expression that returns a string.</td>
</tr>
</tbody>
</table>

**Considerations:**

`trim(null)` returns `null`

**Query**

```
RETURN trim(  '   hello   '  )
```

**Table 318. Result**

<table>
<thead>
<tr>
<th><code>trim(  '   hello   '  )</code></th>
<th>&quot;hello&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

### 10.9. Temporal functions - instant types

_Cypher provides functions allowing for the creation and manipulation of values for each temporal type — Date, Time, LocalTime, DateTime, and LocalDateTime._

**See also** Temporal (Date/Time) values and Temporal operators.

### 10.9.1. Temporal instant types

_An introduction to temporal instant types, including descriptions of creation functions, clocks, and truncation._

**An overview of temporal instant type creation**

Each function bears the same name as the type, and construct the type they correspond to in one of four ways:

- Capturing the current time
- Composing the components of the type
- Parsing a string representation of the temporal value
- Selecting and composing components from another temporal value by
  - either combining temporal values (such as combining a Date with a Time to create a DateTime), or
  - selecting parts from a temporal value (such as selecting the Date from a DateTime); the extractors — groups of components which can be selected — are:
date — contains all components for a Date (conceptually year, month and day).

- time — contains all components for a Time (hour, minute, second, and sub-seconds; namely millisecond, microsecond and nanosecond). If the type being created and the type from which the time component is being selected both contain timezone (and a timezone is not explicitly specified) the timezone is also selected.

- datetime — selects all components, and is useful for overriding specific components. Analogously to time, if the type being created and the type from which the time component is being selected both contain timezone (and a timezone is not explicitly specified) the timezone is also selected.

○ In effect, this allows for the conversion between different temporal types, and allowing for ‘missing’ components to be specified.

Table 319. Temporal instant type creation functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Date</th>
<th>Time</th>
<th>LocalTime</th>
<th>DateTime</th>
<th>LocalDateTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Getting the current value</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Creating a calendar-based (Year-Month-Day) value</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Creating a week-based (Year-Week-Day) value</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Creating a quarter-based (Year-Quarter-Day) value</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Creating an ordinal (Year-Day) value</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Creating a value from time components</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Creating a value from other temporal values using extractors (i.e. converting between different types)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Creating a value from a string</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Creating a value from a timestamp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
Controlling which clock to use

The functions which create temporal instant values based on the current instant use the `statement` clock as default. However, there are three different clocks available for more fine-grained control:

- **transaction**: The same instant is produced for each invocation within the same transaction. A different time may be produced for different transactions.
- **statement**: The same instant is produced for each invocation within the same statement. A different time may be produced for different statements within the same transaction.
- **realtime**: The instant produced will be the live clock of the system.

The following table lists the different sub-functions for specifying the clock to be used when creating the current temporal instant value:

<table>
<thead>
<tr>
<th>Type</th>
<th>default</th>
<th>transaction</th>
<th>statement</th>
<th>realtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>date()</td>
<td>date.transaction()</td>
<td>date.statement()</td>
<td>date.realtime()</td>
</tr>
<tr>
<td>Time</td>
<td>time()</td>
<td>time.transaction()</td>
<td>time.statement()</td>
<td>time.realtime()</td>
</tr>
<tr>
<td>LocalTime</td>
<td>localtime()</td>
<td>localtime.transaction()</td>
<td>localtime.statement()</td>
<td>localtime.realtime()</td>
</tr>
<tr>
<td>DateTime</td>
<td>datetime()</td>
<td>datetime.transaction()</td>
<td>datetime.statement()</td>
<td>datetime.realtime()</td>
</tr>
<tr>
<td>LocalDateTime</td>
<td>localdatetime()</td>
<td>localdatetime.transaction()</td>
<td>localdatetime.statement()</td>
<td>localdatetime.realtime()</td>
</tr>
</tbody>
</table>

Truncating temporal values

A temporal instant value can be created by truncating another temporal instant value at the nearest preceding point in time at a specified component boundary (namely, a truncation unit). A temporal instant value created in this way will have all components which are less significant than the specified truncation unit set to their default values.

It is possible to supplement the truncated value by providing a map containing components which are less significant than the truncation unit. This will have the effect of overriding the default values which would otherwise have been set for these less significant components.

The following truncation units are supported:

- **millennium**: Select the temporal instant corresponding to the millennium of the given instant.
- **century**: Select the temporal instant corresponding to the century of the given instant.
- **decade**: Select the temporal instant corresponding to the decade of the given instant.
- **year**: Select the temporal instant corresponding to the year of the given instant.
• **weekYear**: Select the temporal instant corresponding to the first day of the first week of the week-year of the given instant.

• **quarter**: Select the temporal instant corresponding to the quarter of the year of the given instant.

• **month**: Select the temporal instant corresponding to the month of the given instant.

• **week**: Select the temporal instant corresponding to the week of the given instant.

• **day**: Select the temporal instant corresponding to the month of the given instant.

• **hour**: Select the temporal instant corresponding to the hour of the given instant.

• **minute**: Select the temporal instant corresponding to the minute of the given instant.

• **second**: Select the temporal instant corresponding to the second of the given instant.

• **millisecond**: Select the temporal instant corresponding to the millisecond of the given instant.

• **microsecond**: Select the temporal instant corresponding to the microsecond of the given instant.

The following table lists the supported truncation units and the corresponding sub-functions:

<table>
<thead>
<tr>
<th>Truncation unit</th>
<th>Date</th>
<th>Time</th>
<th>LocalTime</th>
<th>DateTime</th>
<th>LocalDateTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>millennium</td>
<td>date.truncate('millennium', input)</td>
<td></td>
<td></td>
<td>datetime.truncate('millennium', input)</td>
<td>localdatetime.truncate('millennium', input)</td>
</tr>
<tr>
<td>century</td>
<td>date.truncate('century', input)</td>
<td></td>
<td></td>
<td>datetime.truncate('century', input)</td>
<td>localdatetime.truncate('century', input)</td>
</tr>
<tr>
<td>decade</td>
<td>date.truncate('decade', input)</td>
<td></td>
<td></td>
<td>datetime.truncate('decade', input)</td>
<td>localdatetime.truncate('decade', input)</td>
</tr>
<tr>
<td>year</td>
<td>date.truncate('year', input)</td>
<td></td>
<td></td>
<td>datetime.truncate('year', input)</td>
<td>localdatetime.truncate('year', input)</td>
</tr>
<tr>
<td>weekYear</td>
<td>date.truncate('weekYear', input)</td>
<td></td>
<td></td>
<td>datetime.truncate('weekYear', input)</td>
<td>localdatetime.truncate('weekYear', input)</td>
</tr>
<tr>
<td>quarter</td>
<td>date.truncate('quarter', input)</td>
<td></td>
<td></td>
<td>datetime.truncate('quarter', input)</td>
<td>localdatetime.truncate('quarter', input)</td>
</tr>
<tr>
<td>month</td>
<td>date.truncate('month', input)</td>
<td></td>
<td></td>
<td>datetime.truncate('month', input)</td>
<td>localdatetime.truncate('month', input)</td>
</tr>
<tr>
<td>week</td>
<td>date.truncate('week', input)</td>
<td>time.truncate('day', input)</td>
<td>localtime.truncate('day', input)</td>
<td>datetime.truncate('day', input)</td>
<td>localdatetime.truncate('day', input)</td>
</tr>
<tr>
<td>day</td>
<td>date.truncate('day', input)</td>
<td></td>
<td>localtime.truncate('day', input)</td>
<td>datetime.truncate('day', input)</td>
<td>localdatetime.truncate('day', input)</td>
</tr>
<tr>
<td>hour</td>
<td></td>
<td>time.truncate('hour', input)</td>
<td>localtime.truncate('hour', input)</td>
<td>datetime.truncate('hour', input)</td>
<td>localdatetime.truncate('hour', input)</td>
</tr>
<tr>
<td>minute</td>
<td></td>
<td>time.truncate('minute', input)</td>
<td>localtime.truncate('minute', input)</td>
<td>datetime.truncate('minute', input)</td>
<td>localdatetime.truncate('minute', input)</td>
</tr>
<tr>
<td>second</td>
<td></td>
<td>time.truncate('second', input)</td>
<td>localtime.truncate('second', input)</td>
<td>datetime.truncate('second', input)</td>
<td>localdatetime.truncate('second', input)</td>
</tr>
</tbody>
</table>
10.9.2. Date: date()

Details for using the date() function.

- Getting the current Date
- Creating a calendar (Year-Month-Day) Date
- Creating a week (Year-Week-Day) Date
- Creating a quarter (Year-Quarter-Day) Date
- Creating an ordinal (Year-Day) Date
- Creating a Date from a string
- Creating a Date using other temporal values as components
- Truncating a Date

Getting the current Date

date() returns the current Date value. If no time zone parameter is specified, the local time zone will be used.

Syntax: date([[timezone]])

Returns:

A Date.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>timezone</td>
<td>A string expression that represents the time zone</td>
</tr>
</tbody>
</table>

Considerations:

If no parameters are provided, date() must be invoked (date() is invalid).
The current date is returned.

Table 320. Result

currentDate

<table>
<thead>
<tr>
<th>2022-08-09</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

The current date in California is returned.

Table 321. Result

currentDateInLA

<table>
<thead>
<tr>
<th>2022-08-09</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

date.transaction()

date.transaction() returns the current Date value using the transaction clock. This value will be the same for each invocation within the same transaction. However, a different value may be produced for different transactions.

**Syntax:** `date.transaction(){[timezone]}

**Returns:**

A Date.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timezone</td>
<td>A string expression that represents the timezone</td>
</tr>
</tbody>
</table>

Query

```
RETURN date() AS currentDate
```

Table 322. Result

```
<table>
<thead>
<tr>
<th>2022-08-09</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>
```
**date.statement()**

`date.statement()` returns the current Date value using the *statement* clock. This value will be the same for each invocation within the same statement. However, a different value may be produced for different statements within the same transaction.

**Syntax:** `date.statement()`

**Returns:** A Date.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timezone</td>
<td>A string expression that represents the time zone</td>
</tr>
</tbody>
</table>

**Query**

```sql
RETURN date.statement() AS currentDate
```

**Table 323. Result**

<table>
<thead>
<tr>
<th>currentDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>2022-08-09</td>
</tr>
</tbody>
</table>

**date.realtime()**

`date.realtime()` returns the current Date value using the *realtime* clock. This value will be the live clock of the system.

**Syntax:** `date.realtime([timezone])`

**Returns:** A Date.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timezone</td>
<td>A string expression that represents the time zone</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>timezone</td>
<td>A string expression that represents the time zone</td>
</tr>
</tbody>
</table>

Query

```plaintext
RETURN date.realtime() AS currentDate
```

Table 324. Result

<table>
<thead>
<tr>
<th>currentDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>2022-08-09</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

Query

```plaintext
RETURN date.realtime('America/Los Angeles') AS currentDateInLA
```

Table 325. Result

<table>
<thead>
<tr>
<th>currentDateInLA</th>
</tr>
</thead>
<tbody>
<tr>
<td>2022-08-09</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

Creating a calendar (Year-Month-Day) Date

date() returns a Date value with the specified year, month and day component values.

Syntax: `date({year [, month, day]})`

Returns:

A Date.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>year</td>
<td>An expression consisting of at least four digits that specifies the year.</td>
</tr>
<tr>
<td>month</td>
<td>An integer between 1 and 12 that specifies the month.</td>
</tr>
<tr>
<td>day</td>
<td>An integer between 1 and 31 that specifies the day of the month.</td>
</tr>
</tbody>
</table>

Considerations:

The day of the month component will default to 1 if day is omitted.
The month component will default to 1 if month is omitted.

If month is omitted, day must also be omitted.

Query

```sql
UNWIND [ 
    date({year:1984, month:10, day:11}),
    date({year:1984, month:10}),
    date({year:1984})
] as theDate
RETURN theDate
```

Table 326. Result

<table>
<thead>
<tr>
<th>theDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984-10-11</td>
</tr>
<tr>
<td>1984-10-01</td>
</tr>
<tr>
<td>1984-01-01</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>year</td>
<td>An expression consisting of at least four digits that specifies the year.</td>
</tr>
<tr>
<td>week</td>
<td>An integer between 1 and 53 that specifies the week.</td>
</tr>
<tr>
<td>dayOfWeek</td>
<td>An integer between 1 and 7 that specifies the day of the week.</td>
</tr>
</tbody>
</table>

Considerations:

The day of the week component will default to 1 if dayOfWeek is omitted.

The week component will default to 1 if week is omitted.

If week is omitted, dayOfWeek must also be omitted.
Query

```plaintext
UNWIND [
  date({year: 1984, week: 10, dayOfWeek: 3}),
  date({year: 1984, week: 10}),
  date({year: 1984})
] as theDate
RETURN theDate
```

Table 327. Result

<table>
<thead>
<tr>
<th>theDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984-03-07</td>
</tr>
<tr>
<td>1984-03-05</td>
</tr>
<tr>
<td>1984-01-01</td>
</tr>
</tbody>
</table>

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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>year</td>
<td>An expression consisting of at least four digits that specifies the year.</td>
</tr>
<tr>
<td>quarter</td>
<td>An integer between 1 and 4 that specifies the quarter.</td>
</tr>
<tr>
<td>dayOfQuarter</td>
<td>An integer between 1 and 92 that specifies the day of the quarter.</td>
</tr>
</tbody>
</table>

**Considerations:**

- The day of the quarter component will default to 1 if `dayOfQuarter` is omitted.
- The quarter component will default to 1 if `quarter` is omitted.
- If `quarter` is omitted, `dayOfQuarter` must also be omitted.
Query

```csharp
UNWIND [date({year: 1984, quarter: 3, dayOfQuarter: 45}),
       date({year: 1984, quarter: 3}),
       date({year: 1984})]
       as theDate
RETURN theDate
```

Table 328. Result

<table>
<thead>
<tr>
<th>theDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984-08-14</td>
</tr>
<tr>
<td>1984-07-01</td>
</tr>
<tr>
<td>1984-01-01</td>
</tr>
</tbody>
</table>

Rows: 3

Creating an ordinal (Year-Day) Date

date() returns a Date value with the specified year and ordinalDay component values.

Syntax: `date({year [, ordinalDay]})`

Returns:
A Date.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of</td>
<td></td>
</tr>
<tr>
<td>the following:</td>
<td></td>
</tr>
<tr>
<td>year</td>
<td>An expression consisting of at least four digits that specifies the year.</td>
</tr>
<tr>
<td>ordinalDay</td>
<td>An integer between 1 and 366 that specifies the ordinal day of the year.</td>
</tr>
</tbody>
</table>

Considerations:

The ordinal day of the year component will default to 1 if `ordinalDay` is omitted.

Query

```csharp
UNWIND [date({year: 1984, ordinalDay: 202}),
        date({year: 1984})]
       as theDate
RETURN theDate
```

The date corresponding to 11 February 1984 is returned.

Table 329. Result
Creating a Date from a string

date() returns the Date value obtained by parsing a string representation of a temporal value.

Syntax: date(temporalValue)

Returns:
A Date.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>temporalValue</td>
<td>A string representing a temporal value.</td>
</tr>
</tbody>
</table>

Considerations:

temporalValue must comply with the format defined for dates.
temporalValue must denote a valid date; i.e. a temporalValue denoting 30 February 2001 is invalid.
date(null) returns null.

Query

```
UNWIND [ date('2015-07-21'),
        date('2015-07'),
        date('201507'),
        date('2015-W30-2'),
        date('2015202'),
        date('2015')
] as theDate
RETURN theDate
```

Table 330. Result

<table>
<thead>
<tr>
<th>theDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015-07-21</td>
</tr>
<tr>
<td>2015-07-01</td>
</tr>
<tr>
<td>2015-07-01</td>
</tr>
<tr>
<td>2015-07-21</td>
</tr>
<tr>
<td>2015-07-21</td>
</tr>
<tr>
<td>2015-01-01</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>date</td>
<td>A Date value.</td>
</tr>
<tr>
<td>year</td>
<td>An expression consisting of at least four digits that specifies the year.</td>
</tr>
<tr>
<td>month</td>
<td>An integer between 1 and 12 that specifies the month.</td>
</tr>
<tr>
<td>day</td>
<td>An integer between 1 and 31 that specifies the day of the month.</td>
</tr>
<tr>
<td>week</td>
<td>An integer between 1 and 53 that specifies the week.</td>
</tr>
<tr>
<td>dayOfWeek</td>
<td>An integer between 1 and 7 that specifies the day of the week.</td>
</tr>
<tr>
<td>quarter</td>
<td>An integer between 1 and 4 that specifies the quarter.</td>
</tr>
<tr>
<td>dayOfQuarter</td>
<td>An integer between 1 and 92 that specifies the day of the quarter.</td>
</tr>
<tr>
<td>ordinalDay</td>
<td>An integer between 1 and 366 that specifies the ordinal day of the year.</td>
</tr>
</tbody>
</table>

Considerations:

If any of the optional parameters are provided, these will override the corresponding components of date.

date(dd) may be written instead of date({date: dd}).
Query

```plaintext
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>
<thead>
<tr>
<th>dateOnly</th>
<th>dateDay</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984-11-11</td>
<td>1984-11-28</td>
</tr>
<tr>
<td>1984-11-11</td>
<td>1984-11-28</td>
</tr>
<tr>
<td>1984-11-11</td>
<td>1984-11-28</td>
</tr>
</tbody>
</table>

Rows: 3

### Truncating a Date

date.truncate() returns the Date value obtained by truncating a specified temporal instant value at the nearest preceding point in time at the specified component boundary (which is denoted by the truncation unit passed as a parameter to the function). In other words, the Date returned will have all components that are less significant than the specified truncation unit set to their default values.

It is possible to supplement the truncated value by providing a map containing components which are less significant than the truncation unit. This will have the effect of overriding the default values which would otherwise have been set for these less significant components. For example, day — with some value x — may be provided when the truncation unit is year in order to ensure the returned value has the day set to x instead of the default day (which is 1).

**Syntax:** date.truncate(unit [, temporalInstantValue [, mapOfComponents ]])

**Returns:**

A Date.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unit</td>
<td>A string expression evaluating to one of the following: millennium, century, decade, year, weekYear, quarter, month, week, day.</td>
</tr>
<tr>
<td>temporalInstantValue</td>
<td>An expression of one of the following types: {DateTime, LocalDateTime, Date}.</td>
</tr>
<tr>
<td>mapOfComponents</td>
<td>An expression evaluating to a map containing components less significant than unit.</td>
</tr>
</tbody>
</table>

**Considerations:**
Any component that is provided in `mapOfComponents` must be less significant than `unit`; i.e. if `unit` is 'day', `mapOfComponents` cannot contain information pertaining to a month.

Any component that is not contained in `mapOfComponents` and which is less significant than `unit` will be set to its minimal value.

If `mapOfComponents` is not provided, all components of the returned value which are less significant than `unit` will be set to their default values.

If `temporalInstantValue` is not provided, it will be set to the current date, i.e. `date.truncate(unit)` is equivalent of `date.truncate(unit, date())`.

**Query**

```sql
WITH datetime({year:2017, month:11, day:11, hour:12, minute:31, second:14, nanosecond: 645876123, timezone: '+01:00'}) AS d
RETURN date.truncate('millennium', d) AS truncMillenium,
     date.truncate('century', d) AS truncCentury,
     date.truncate('decade', d) AS truncDecade,
     date.truncate('year', d, {day:5}) AS truncYear,
     date.truncate('weekYear', d) AS truncWeekYear,
     date.truncate('quarter', d) AS truncQuarter,
     date.truncate('month', d) AS truncMonth,
     date.truncate('week', d, {dayOfWeek:2}) AS truncWeek,
     date.truncate('day', d) AS truncDay
```

**Table 332. Result**

<table>
<thead>
<tr>
<th>truncMillenium</th>
<th>truncCentury</th>
<th>truncDecade</th>
<th>truncYear</th>
<th>truncWeekYear</th>
<th>truncQuarter</th>
<th>truncMonth</th>
<th>truncWeek</th>
<th>truncDay</th>
</tr>
</thead>
</table>

**10.9.3. DateTime: `datetime()`**

Details for using the `datetime()` function.

- Getting the current `DateTime`
- Creating a calendar (Year-Month-Day) `DateTime`
- Creating a week (Year-Week-Day) `DateTime`
- Creating a quarter (Year-Quarter-Day) `DateTime`
- Creating an ordinal (Year-Day) `DateTime`
- Creating a `DateTime` from a string
- Creating a `DateTime` using other temporal values as components
- Creating a `DateTime` from a timestamp
- Truncating a `DateTime`
Getting the current DateTime

datetime() returns the current DateTime value. If no time zone parameter is specified, the default time zone will be used.

Syntax: datetime([timezone])

Returns:

A DateTime.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>timezone</td>
<td>A string expression that represents the time zone</td>
</tr>
</tbody>
</table>

Considerations:

If no parameters are provided, datetime() must be invoked (datetime() is invalid).

Query

```sql
RETURN datetime() AS currentDateTime
```

The current date and time using the local time zone is returned.

Table 333. Result

<table>
<thead>
<tr>
<th>currentDateTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>2022-08-09T14:22:00.706Z</td>
</tr>
</tbody>
</table>

Rows: 1

Query

```sql
RETURN datetime({timezone: 'America/Los Angeles'}) AS currentDateTimeInLA
```

The current date and time of day in California is returned.

Table 334. Result

<table>
<thead>
<tr>
<th>currentDateTimeInLA</th>
</tr>
</thead>
<tbody>
<tr>
<td>2022-08-09T07:22:00.722-07:00[America/Los_Angeles]</td>
</tr>
</tbody>
</table>

Rows: 1


datetime.transaction()

datetime.transaction() returns the current DateTime value using the transaction clock. This value will be the same for each invocation within the same transaction. However, a different value may be produced for different transactions.

Syntax: datetime.transaction([{'timezone']})

Returns:

A DateTime.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timezone</td>
<td>A string expression that represents the time zone</td>
</tr>
</tbody>
</table>

Query

```
RETURN datetime.transaction() AS currentDateTime
```

Table 335. Result

<table>
<thead>
<tr>
<th>currentDateTime</th>
<th>2022-08-09T14:22:00.737Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

Query

```
RETURN datetime.transaction('America/Los Angeles') AS currentDateTimeInLA
```

Table 336. Result

<table>
<thead>
<tr>
<th>currentDateTimeInLA</th>
<th>2022-08-09T07:22:00.750-07:00[America/Los_Angeles]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

datetime.statement()

datetime.statement() returns the current DateTime value using the statement clock. This value will be the same for each invocation within the same statement. However, a different value may be produced for different statements within the same transaction.

Syntax: datetime.statement([{'timezone']})

Returns:
A DateTime.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timezone</td>
<td>A string expression that represents the timezone</td>
</tr>
</tbody>
</table>

Query

```
RETURN datetime.statement() AS currentDateTime
```

Table 337. Result

<table>
<thead>
<tr>
<th>currentDateTime</th>
<th>2022-08-09T14:22:00.764Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows</td>
<td>1</td>
</tr>
</tbody>
</table>

datetime.realtime()

datetime.realtime() returns the current DateTime value using the realtime clock. This value will be the live clock of the system.

Syntax: `datetime.realtime([timezone])`

Returns:

A DateTime.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timezone</td>
<td>A string expression that represents the timezone</td>
</tr>
</tbody>
</table>

Query

```
RETURN datetime.realtime() AS currentDateTime
```

Table 338. Result

<table>
<thead>
<tr>
<th>currentDateTime</th>
<th>2022-08-09T14:22:00.789649Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows</td>
<td>1</td>
</tr>
</tbody>
</table>

Creating a calendar (Year-Month-Day) DateTime

datetime() returns a DateTime value with the specified year, month, day, hour, minute, second,
millisecond, microsecond, nanosecond and timezone component values.

Syntax: \texttt{datetime(year [, month, day, hour, minute, second, millisecond, microsecond, nanosecond, timezone])}

Returns:

A DateTime.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>year</td>
<td>An expression consisting of at least four digits that specifies the year.</td>
</tr>
<tr>
<td>month</td>
<td>An integer between 1 and 12 that specifies the month.</td>
</tr>
<tr>
<td>day</td>
<td>An integer between 1 and 31 that specifies the day of the month.</td>
</tr>
<tr>
<td>hour</td>
<td>An integer between 0 and 23 that specifies the hour of the day.</td>
</tr>
<tr>
<td>minute</td>
<td>An integer between 0 and 59 that specifies the number of minutes.</td>
</tr>
<tr>
<td>second</td>
<td>An integer between 0 and 59 that specifies the number of seconds.</td>
</tr>
<tr>
<td>millisecond</td>
<td>An integer between 0 and 999 that specifies the number of milliseconds.</td>
</tr>
<tr>
<td>microsecond</td>
<td>An integer between 0 and 999,999 that specifies the number of microseconds.</td>
</tr>
<tr>
<td>nanosecond</td>
<td>An integer between 0 and 999,999,999 that specifies the number of nanoseconds.</td>
</tr>
<tr>
<td>timezone</td>
<td>An expression that specifies the time zone.</td>
</tr>
</tbody>
</table>

Considerations:

- The month component will default to 1 if month is omitted.
- The day of the month component will default to 1 if day is omitted.
- The hour component will default to 0 if hour is omitted.
- The minute component will default to 0 if minute is omitted.
- The second component will default to 0 if second is omitted.
- Any missing millisecond, microsecond or nanosecond values will default to 0.
- The timezone component will default to the configured default time zone if timezone is omitted.
If millisecond, microsecond and nanosecond are given in combination (as part of the same set of parameters), the individual values must be in the range 0 to 999.

The least significant components in the set year, month, day, hour, minute, and second may be omitted; i.e. it is possible to specify only year, month and day, but specifying year, month, day and minute is not permitted.

One or more of millisecond, microsecond and nanosecond can only be specified as long as second is also specified.

Query

```sql
UNWIND [
  datetime({year: 1984, month: 10, day: 11, hour: 12, minute: 31, second: 14, millisecond: 645, timezone: '+01:00'}),
  datetime({year: 1984, month: 10, day: 11, hour: 12, minute: 31, second: 14, timezone: '+01:00'}),
  datetime({year: 1984, month: 10, day: 11, hour: 12, minute: 31, second: 14, timezone: 'Europe/Stockholm'}),
  datetime({year: 1984, month: 10, day: 11, hour: 12, minute: 31, second: 14, timezone: '+01:00'}, 'Europe/Stockholm'))
] as theDate
RETURN theDate
```

Table 339. Result

<table>
<thead>
<tr>
<th>theDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984-10-11T12:31:14.645+01:00</td>
</tr>
<tr>
<td>1984-10-11T12:31:14+01:00</td>
</tr>
<tr>
<td>1984-10-11T12:31:14Z</td>
</tr>
<tr>
<td>1984-10-11T12:31+01:00[Europe/Stockholm]</td>
</tr>
<tr>
<td>1984-10-11T12:00+01:00</td>
</tr>
<tr>
<td>1984-10-11T00:00+01:00[Europe/Stockholm]</td>
</tr>
</tbody>
</table>

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:
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>year</td>
<td>An expression consisting of at least four digits that specifies the year.</td>
</tr>
<tr>
<td>week</td>
<td>An integer between 1 and 53 that specifies the week.</td>
</tr>
<tr>
<td>dayOfWeek</td>
<td>An integer between 1 and 7 that specifies the day of the week.</td>
</tr>
<tr>
<td>hour</td>
<td>An integer between 0 and 23 that specifies the hour of the day.</td>
</tr>
<tr>
<td>minute</td>
<td>An integer between 0 and 59 that specifies the number of minutes.</td>
</tr>
<tr>
<td>second</td>
<td>An integer between 0 and 59 that specifies the number of seconds.</td>
</tr>
<tr>
<td>millisecond</td>
<td>An integer between 0 and 999 that specifies the number of milliseconds.</td>
</tr>
<tr>
<td>microsecond</td>
<td>An integer between 0 and 999,999 that specifies the number of microseconds.</td>
</tr>
<tr>
<td>nanosecond</td>
<td>An integer between 0 and 999,999,999 that specifies the number of nanoseconds.</td>
</tr>
<tr>
<td>timezone</td>
<td>An expression that specifies the time zone.</td>
</tr>
</tbody>
</table>

**Considerations:**

- The week component will default to 1 if `week` is omitted.
- The day of the week component will default to 1 if `dayOfWeek` is omitted.
- The hour component will default to 0 if `hour` is omitted.
- The minute component will default to 0 if `minute` is omitted.
- The second component will default to 0 if `second` is omitted.
- Any missing `millisecond`, `microsecond` or `nanosecond` values will default to 0.
- The timezone component will default to the configured default time zone if `timezone` is omitted.
- If `millisecond`, `microsecond` and `nanosecond` are given in combination (as part of the same set of parameters), the individual values must be in the range 0 to 999.
- The least significant components in the set `year`, `week`, `dayOfWeek`, `hour`, `minute`, and `second` may be omitted; i.e. it is possible to specify only `year`, `week` and `dayOfWeek`, but specifying `year`, `week`, `dayOfWeek` and `minute` is not permitted.
- One or more of `millisecond`, `microsecond` and `nanosecond` can only be specified as long as `second` is also specified.
Query

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, 
    millisecond: 645, 
    timezone: '+01:00'
  }),
  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, 
    timezone: 'Europe/Stockholm'
}),
  datetime({
    year: 1984, 
    week: 10, 
    dayOfWeek: 3, 
    hour: 12, 
    minute: 31, 
    second: 14, 
    timezone: 'Europe/Stockholm'
})
] as theDate
RETURN theDate

Table 340. Result

<table>
<thead>
<tr>
<th>theDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984-03-07T12:31:14.645Z</td>
</tr>
<tr>
<td>1984-03-07T12:31:14.645876+01:00</td>
</tr>
<tr>
<td>1984-03-07T12:31:14.645876123+01:00[Europe/Stockholm]</td>
</tr>
<tr>
<td>1984-03-07T12:31:14+01:00[Europe/Stockholm]</td>
</tr>
<tr>
<td>1984-03-07T12:31:14Z</td>
</tr>
<tr>
<td>1984-03-07T12:00+01:00</td>
</tr>
<tr>
<td>1984-03-07T00:00+01:00[Europe/Stockholm]</td>
</tr>
</tbody>
</table>

Rows: 7

Creating a quarter (Year-Quarter-Day) DateTime

datetime() returns a DateTime value with the specified year, quarter, dayOfQuarter, hour, minute, second, millisecond, microsecond, nanosecond and timezone component values.

Syntax: datetime({year [, quarter, dayOfQuarter, hour, minute, second, millisecond, microsecond, nanosecond, timezone]])

Returns:

A DateTime.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>year</td>
<td>An expression consisting of at least four digits that specifies the year.</td>
</tr>
<tr>
<td>quarter</td>
<td>An integer between 1 and 4 that specifies the quarter.</td>
</tr>
<tr>
<td>dayOfQuarter</td>
<td>An integer between 1 and 92 that specifies the day of the quarter.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>hour</td>
<td>An integer between 0 and 23 that specifies the hour of the day.</td>
</tr>
<tr>
<td>minute</td>
<td>An integer between 0 and 59 that specifies the number of minutes.</td>
</tr>
<tr>
<td>second</td>
<td>An integer between 0 and 59 that specifies the number of seconds.</td>
</tr>
<tr>
<td>millisecond</td>
<td>An integer between 0 and 999 that specifies the number of milliseconds.</td>
</tr>
<tr>
<td>microsecond</td>
<td>An integer between 0 and 999,999 that specifies the number of microseconds.</td>
</tr>
<tr>
<td>nanosecond</td>
<td>An integer between 0 and 999,999,999 that specifies the number of nanoseconds.</td>
</tr>
<tr>
<td>timezone</td>
<td>An expression that specifies the time zone.</td>
</tr>
</tbody>
</table>

**Considerations:**

The quarter component will default to 1 if `quarter` is omitted.

The day of the quarter component will default to 1 if `dayOfQuarter` is omitted.

The hour component will default to 0 if `hour` is omitted.

The minute component will default to 0 if `minute` is omitted.

The second component will default to 0 if `second` is omitted.

Any missing `millisecond`, `microsecond` or `nanosecond` values will default to 0.

The timezone component will default to the configured default time zone if `timezone` is omitted.

If `millisecond`, `microsecond` and `nanosecond` are given in combination (as part of the same set of parameters), the individual values must be in the range 0 to 999.

The least significant components in the set `year`, `quarter`, `dayOfQuarter`, `hour`, `minute`, and `second` may be omitted; i.e. it is possible to specify only `year`, `quarter` and `dayOfQuarter`, but specifying `year`, `quarter`, `dayOfQuarter` and `minute` is not permitted.

One or more of `millisecond`, `microsecond` and `nanosecond` can only be specified as long as `second` is also specified.

**Query**

```sql
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 341. Result**

<table>
<thead>
<tr>
<th>theDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984-08-14T12:31:14.645876Z</td>
</tr>
</tbody>
</table>
Creating an ordinal (Year-Day) DateTime

datetime() returns a DateTime value with the specified year, ordinalDay, hour, minute, second, millisecond, microsecond, nanosecond and timezone component values.

Syntax: datetime({year [, ordinalDay, hour, minute, second, millisecond, microsecond, nanosecond, timezone]})

Returns:

A DateTime.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>year</td>
<td>An expression consisting of at least four digits that specifies the year.</td>
</tr>
<tr>
<td>ordinalDay</td>
<td>An integer between 1 and 366 that specifies the ordinal day of the year.</td>
</tr>
<tr>
<td>hour</td>
<td>An integer between 0 and 23 that specifies the hour of the day.</td>
</tr>
<tr>
<td>minute</td>
<td>An integer between 0 and 59 that specifies the number of minutes.</td>
</tr>
<tr>
<td>second</td>
<td>An integer between 0 and 59 that specifies the number of seconds.</td>
</tr>
<tr>
<td>millisecond</td>
<td>An integer between 0 and 999 that specifies the number of milliseconds.</td>
</tr>
<tr>
<td>microsecond</td>
<td>An integer between 0 and 999,999 that specifies the number of microseconds.</td>
</tr>
<tr>
<td>nanosecond</td>
<td>An integer between 0 and 999,999,999 that specifies the number of nanoseconds.</td>
</tr>
<tr>
<td>timezone</td>
<td>An expression that specifies the time zone.</td>
</tr>
</tbody>
</table>

Considerations:

The ordinal day of the year component will default to 1 if ordinalDay is omitted.
The hour component will default to 0 if `hour` is omitted.

The minute component will default to 0 if `minute` is omitted.

The second component will default to 0 if `second` is omitted.

Any missing `millisecond`, `microsecond` or `nanosecond` values will default to 0.

The timezone component will default to the configured default time zone if `timezone` is omitted.

If `millisecond`, `microsecond` and `nanosecond` are given in combination (as part of the same set of parameters), the individual values must be in the range 0 to 999.

The least significant components in the set `year`, `ordinalDay`, `hour`, `minute`, and `second` may be omitted; i.e. it is possible to specify only `year` and `ordinalDay`, but specifying `year`, `ordinalDay` and `minute` is not permitted.

One or more of `millisecond`, `microsecond` and `nanosecond` can only be specified as long as `second` is also specified.

Query

```plaintext
UNWIND [
    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 342. Result

<table>
<thead>
<tr>
<th>theDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984-07-20T12:31:14+01:00</td>
</tr>
<tr>
<td>1984-07-20T00:00+02:00[Europe/Stockholm]</td>
</tr>
<tr>
<td>1984-07-20T00:00Z</td>
</tr>
</tbody>
</table>

Rows: 4

Creating a `DateTime` from a string

datetime() returns the `DateTime` value obtained by parsing a string representation of a temporal value.

Syntax: `datetime(temporalValue)`

Returns:

A `DateTime`.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>temporalValue</td>
<td>A string representing a temporal value.</td>
</tr>
</tbody>
</table>

Considerations:
temporalValue must comply with the format defined for dates, times and time zones.

The timezone component will default to the configured default time zone if it is omitted.

temporalValue must denote a valid date and time; i.e. a temporalValue denoting 30 February 2001 is invalid.

datetime(null) returns null.

Query

```sql
UNWIND [
  datetime('2015-07-21T21:40:32.142+0100'),
  datetime('2015-W30-2T214032.142+0100'),
  datetime('20150721T21:40:32-01:30'),
  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 343. Result

<table>
<thead>
<tr>
<th>theDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015-07-21T21:40:32.142+01:00</td>
</tr>
<tr>
<td>2015-07-21T21:40:32.142Z</td>
</tr>
<tr>
<td>2015-01-01T21:40:32-01:00</td>
</tr>
<tr>
<td>2015-07-21T21:40-01:30</td>
</tr>
<tr>
<td>2015-07-20T21:40-02:00</td>
</tr>
<tr>
<td>2015-07-21T21:00+18:00</td>
</tr>
<tr>
<td>2015-07-21T21:40:32.142+01:00[Europe/London]</td>
</tr>
<tr>
<td>2015-07-21T21:40:32.142-04:00[America/New_York]</td>
</tr>
</tbody>
</table>

Rows: 8

Creating a DateTime using other temporal values as components

datetime() returns the DateTime value obtained by selecting and composing components from another temporal value. In essence, this allows a Date, LocalDateTime, Time or LocalTime value to be converted to a DateTime, and for "missing" components to be provided.

Syntax: datetime({datetime [, year, …, timezone]}) | datetime({date [, year, …, timezone]}) | datetime({time [, year, …, timezone]}) | datetime({date, time [, year, …, timezone]})

Returns:

A DateTime.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>datetime</td>
<td>A DateTime value.</td>
</tr>
<tr>
<td>date</td>
<td>A Date value.</td>
</tr>
<tr>
<td>time</td>
<td>A Time value.</td>
</tr>
<tr>
<td>year</td>
<td>An expression consisting of at least four digits that specifies the year.</td>
</tr>
<tr>
<td>month</td>
<td>An integer between 1 and 12 that specifies the month.</td>
</tr>
<tr>
<td>day</td>
<td>An integer between 1 and 31 that specifies the day of the month.</td>
</tr>
<tr>
<td>week</td>
<td>An integer between 1 and 53 that specifies the week.</td>
</tr>
<tr>
<td>dayOfWeek</td>
<td>An integer between 1 and 7 that specifies the day of the week.</td>
</tr>
<tr>
<td>quarter</td>
<td>An integer between 1 and 4 that specifies the quarter.</td>
</tr>
<tr>
<td>dayOfQuarter</td>
<td>An integer between 1 and 92 that specifies the day of the quarter.</td>
</tr>
<tr>
<td>ordinalDay</td>
<td>An integer between 1 and 366 that specifies the ordinal day of the year.</td>
</tr>
<tr>
<td>hour</td>
<td>An integer between 0 and 23 that specifies the hour of the day.</td>
</tr>
<tr>
<td>minute</td>
<td>An integer between 0 and 59 that specifies the number of minutes.</td>
</tr>
<tr>
<td>second</td>
<td>An integer between 0 and 59 that specifies the number of seconds.</td>
</tr>
<tr>
<td>millisecond</td>
<td>An integer between 0 and 999 that specifies the number of milliseconds.</td>
</tr>
<tr>
<td>microsecond</td>
<td>An integer between 0 and 999,999 that specifies the number of microseconds.</td>
</tr>
<tr>
<td>nanosecond</td>
<td>An integer between 0 and 999,999,999 that specifies the number of nanoseconds.</td>
</tr>
<tr>
<td>timezone</td>
<td>An expression that specifies the time zone.</td>
</tr>
</tbody>
</table>

**Considerations:**

If any of the optional parameters are provided, these will override the corresponding components of `datetime`, `date` and/or `time`.

`datetime(dd)` may be written instead of `datetime({datetime: dd})`.

Selecting a `Time` or `DateTime` value as the `time` component also selects its time zone. If a `LocalTime` or `LocalDateTime` is selected instead, the default time zone is used. In any case, the time zone can be overridden explicitly.

Selecting a `DateTime` as the `datetime` component and overwriting the time zone will adjust the local time to keep the same point in time.
Selecting a DateTime or Time as the time component and overwriting the time zone will adjust the local time to keep the same point in time.

The following query shows the various usages of `datetime({date [, year, ... , timezone]})`

**Query**

```sql
WITH date({year: 1984, month: 10, day: 11}) AS dd
RETURN datetime({date: dd, hour: 10, minute: 10, second: 10}) AS dateHHMMSS,
    datetime({date: dd, hour: 10, minute: 10, second: 10, timezone: '+05:00'}) AS dateHHMMSSTimezone,
    datetime({date: dd, day: 28, hour: 10, minute: 10, second: 10}) AS dateDDHHMMSSTimezone
```

**Table 344. Result**

<table>
<thead>
<tr>
<th>dateHHMMSS</th>
<th>dateHHMMSSTimezone</th>
<th>dateDDHHMMSS</th>
<th>dateDDHHMMSSTimezone</th>
</tr>
</thead>
</table>

The following query shows the various usages of `datetime({time [, year, ... , timezone]})`

**Query**

```sql
WITH time({hour: 12, minute: 31, second: 14, microsecond: 645676, 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, timezone: 'Pacific/Honolulu'}) AS YYYYMMDDTimeSSTimezone
```

**Table 345. Result**

<table>
<thead>
<tr>
<th>YYYYMMDDTime</th>
<th>YYYYMMDDTimeTimezone</th>
<th>YYYYMMDDTimeSS</th>
<th>YYYYMMDDTimeSSTimezone</th>
</tr>
</thead>
</table>

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**

```sql
WITH date({year: 1984, month: 10, day: 11}) AS dd,
    localtime({hour: 12, minute: 31, second: 14, millisecond: 645}) AS tt
RETURN datetime({date: dd, time: tt}) as dateTime,
    datetime({date: dd, time: tt, timezone: '+05:00'}) AS dateTimeTimezone,
    datetime({date: dd, time: tt, day: 28, second: 42}) AS dateTimeDDSS,
    datetime({date: dd, time: tt, day: 28, second: 42, timezone: 'Pacific/Honolulu'}) AS dateTimeDDSSTimezone
```

**Table 346. Result**

<table>
<thead>
<tr>
<th>dateTime</th>
<th>dateTimeTimezone</th>
<th>dateTimeDDSS</th>
<th>dateTimeDDSSTimezone</th>
</tr>
</thead>
</table>
The following query shows the various usages of `datetime([datetime [, year, …, timezone]])`.

**Query**

```sql
WITH
datetime({
    year: 1984,
    month: 10,
    day: 11,
    hour: 12,
    timezone: 'Europe/Stockholm'
}) AS dd
RETURN
datetime({
    datetime: dd
}) AS dateTime,

datetime({
    datetime: dd,
    timezone: '+05:00'
}) AS dateTimeTimezone,

datetime({
    datetime: dd,
    day: 28,
    second: 42
}) AS dateTimeDDSS,

datetime({
    datetime: dd,
    day: 28,
    second: 42,
    timezone: 'Pacific/Honolulu'
}) AS dateTimeDDSSTimezone
```

**Table 347. Result**

<table>
<thead>
<tr>
<th>dateTime</th>
<th>dateTimeTimezone</th>
<th>dateTimeDDSS</th>
<th>dateTimeDDSSTimezone</th>
</tr>
</thead>
</table>

Creating a **DateTime** from a timestamp

`datetime()` returns the DateTime value at the specified number of seconds or milliseconds from the UNIX epoch in the UTC time zone.

Conversions to other temporal instant types from UNIX epoch representations can be achieved by transforming a DateTime value to one of these types.

**Syntax:** `datetime({'epochSeconds | epochMillis'})`

**Returns:** A DateTime.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>epochSeconds</td>
<td>A numeric value representing the number of seconds from the UNIX epoch in the UTC time zone.</td>
</tr>
<tr>
<td>epochMillis</td>
<td>A numeric value representing the number of milliseconds from the UNIX epoch in the UTC time zone.</td>
</tr>
</tbody>
</table>

**Considerations:**

`epochSeconds/epochMillis` may be used in conjunction with nanosecond
Truncating a DateTime

datetime.truncate() returns the DateTime value obtained by truncating a specified temporal instant value at the nearest preceding point in time at the specified component boundary (which is denoted by the truncation unit passed as a parameter to the function). In other words, the DateTime returned will have all components that are less significant than the specified truncation unit set to their default values.

It is possible to supplement the truncated value by providing a map containing components which are less significant than the truncation unit. This will have the effect of overriding the default values which would otherwise have been set for these less significant components. For example, day — with some value \( x \) — may be provided when the truncation unit is year in order to ensure the returned value has the day set to \( x \) instead of the default day (which is 1).

Syntax: `datetime.truncate(unit [, temporalInstantValue [, mapOfComponents ]])`

Returns:

A DateTime.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>unit</code></td>
<td>A string expression evaluating to one of the following: millennium, century, decade, year, weekYear, quarter, month, week, day, hour, minute, second, millisecond, microsecond.</td>
</tr>
<tr>
<td><code>temporalInstantValue</code></td>
<td>An expression of one of the following types: DateTime, LocalDateTime, Date.</td>
</tr>
</tbody>
</table>
Considerations:

temporalInstantValue cannot be a Date value if unit is one of {hour, minute, second, millisecond, microsecond}.

The time zone of temporalInstantValue may be overridden; for example, datetime.truncate('minute', input, {timezone: '+0200'}). If temporalInstantValue is one of {Time, DateTime} — a value with a time zone — and the time zone is overridden, no time conversion occurs. If temporalInstantValue is one of {LocalDateTime, Date} — a value without a time zone — and the time zone is not overridden, the configured default time zone will be used.

Any component that is provided in mapOfComponents must be less significant than unit; i.e. if unit is 'day', mapOfComponents cannot contain information pertaining to a month.

Any component that is not contained in mapOfComponents and which is less significant than unit will be set to its minimal value.

If mapOfComponents is not provided, all components of the returned value which are less significant than unit will be set to their default values.

If temporalInstantValue is not provided, it will be set to the current date, time and timezone, i.e. datetime.truncate(unit) is equivalent of datetime.truncate(unit, datetime()).

Query

```
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 350. Result

<table>
<thead>
<tr>
<th>truncMillenium</th>
<th>truncYear</th>
<th>truncMonth</th>
<th>truncDay</th>
<th>truncHour</th>
<th>truncSecond</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01T00:00:00+01:00[Europe/Stockholm]</td>
<td>2017-01-05T00:00:00</td>
<td>2017-11-01T00:00:00</td>
<td>2017-11-11T00:00:00.002+03:00</td>
<td>2017-11-11T12:00:00+03:00</td>
<td>2017-11-11T12:31:14+03:00</td>
</tr>
</tbody>
</table>

Rows: 1

10.9.4. LocalDateTime: `localdatetime()`

*Details for using the `localdatetime()` function.*

- Getting the current LocalDateTime
Creating a calendar (Year-Month-Day) LocalDateTime

Creating a week (Year-Week-Day) LocalDateTime

Creating a quarter (Year-Quarter-Day) LocalDateTime

Creating an ordinal (Year-Day) LocalDateTime

Creating a LocalDateTime from a string

Creating a LocalDateTime using other temporal values as components

Truncating a LocalDateTime

Getting the current LocalDateTime

`localdatetime()` returns the current LocalDateTime value. If no time zone parameter is specified, the local time zone will be used.

Syntax: `localdatetime([{'timezone'}])`

Returns:

A LocalDateTime.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>timezone</td>
<td>A string expression that represents the time zone</td>
</tr>
</tbody>
</table>

Considerations:

If no parameters are provided, `localdatetime()` must be invoked (`localdatetime({})` is invalid).

Query

```
RETURN localdatetime() AS now
```

The current local date and time (i.e. in the local time zone) is returned.

Table 351. Result

<table>
<thead>
<tr>
<th>now</th>
</tr>
</thead>
<tbody>
<tr>
<td>2022-08-09T14:22:01.389</td>
</tr>
</tbody>
</table>

Rows: 1

Query

```
RETURN localdatetime({timezone: 'America/Los Angeles'}) AS now
```

The current local date and time in California is returned.

288
localdatetime.transaction()

`localdatetime.transaction()` returns the current `LocalDateTime` value using the `transaction` clock. This value will be the same for each invocation within the same transaction. However, a different value may be produced for different transactions.

**Syntax:** `localdatetime.transaction([timezone])`

**Returns:**

A `LocalDateTime`.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timezone</td>
<td>A string expression that represents the time zone</td>
</tr>
</tbody>
</table>

**Query**

```
RETURN localdatetime.transaction() AS now
```

Table 353. Result

<table>
<thead>
<tr>
<th>now</th>
</tr>
</thead>
<tbody>
<tr>
<td>2022-08-09T14:22:01.422</td>
</tr>
</tbody>
</table>

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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timezone</td>
<td>A string expression that represents the time zone</td>
</tr>
</tbody>
</table>

**Query**

```
RETURN localdatetime.statement() AS now
```

Table 353. Result

<table>
<thead>
<tr>
<th>now</th>
</tr>
</thead>
<tbody>
<tr>
<td>2022-08-09T14:22:01.422</td>
</tr>
</tbody>
</table>

Rows: 1
timezone

A string expression that represents the time zone

Query

RETURN localdatetime.statement() AS now

Table 354. Result

<table>
<thead>
<tr>
<th>now</th>
<th>2022-08-09T14:22:01.437</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

localdatetime.realtime()

localdatetime.realtime() returns the current LocalDateTime value using the realtime clock. This value will be the live clock of the system.

Syntax: localdatetime.realtime([timezone])

Returns:

A LocalDateTime.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timezone</td>
<td>A string expression that represents the time zone</td>
</tr>
</tbody>
</table>

Query

RETURN localdatetime.realtime() AS now

Table 355. Result

<table>
<thead>
<tr>
<th>now</th>
<th>2022-08-09T14:22:01.458418</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

Query

RETURN localdatetime.realtime('America/Los Angeles') AS nowInLA

Table 356. Result

<table>
<thead>
<tr>
<th>nowInLA</th>
<th>2022-08-09T07:22:01.469880</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Creating a calendar (Year-Month-Day) LocalDateTime

`localdatetime()` returns a LocalDateTime value with the specified year, month, day, hour, minute, second, millisecond, microsecond and nanosecond component values.

**Syntax:** `localdatetime([year [, month, day, hour, minute, second, millisecond, microsecond, nanosecond]])`

**Returns:**
A LocalDateTime.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>year</td>
<td>An expression consisting of at least four digits that specifies the year.</td>
</tr>
<tr>
<td>month</td>
<td>An integer between 1 and 12 that specifies the month.</td>
</tr>
<tr>
<td>day</td>
<td>An integer between 1 and 31 that specifies the day of the month.</td>
</tr>
<tr>
<td>hour</td>
<td>An integer between 0 and 23 that specifies the hour of the day.</td>
</tr>
<tr>
<td>minute</td>
<td>An integer between 0 and 59 that specifies the number of minutes.</td>
</tr>
<tr>
<td>second</td>
<td>An integer between 0 and 59 that specifies the number of seconds.</td>
</tr>
<tr>
<td>millisecond</td>
<td>An integer between 0 and 999 that specifies the number of milliseconds.</td>
</tr>
<tr>
<td>microsecond</td>
<td>An integer between 0 and 999,999 that specifies the number of microseconds.</td>
</tr>
<tr>
<td>nanosecond</td>
<td>An integer between 0 and 999,999,999 that specifies the number of nanoseconds.</td>
</tr>
</tbody>
</table>

**Considerations:**

- The month component will default to 1 if `month` is omitted.
- The day of the month component will default to 1 if `day` is omitted.
- The hour component will default to 0 if `hour` is omitted.
- The minute component will default to 0 if `minute` is omitted.
The second component will default to 0 if second is omitted.

Any missing millisecond, microsecond or nanosecond values will default to 0.

If millisecond, microsecond and nanosecond are given in combination (as part of the same set of parameters), the individual values must be in the range 0 to 999.

The least significant components in the set year, month, day, hour, minute, and second may be omitted; i.e. it is possible to specify only year, month and day, but specifying year, month, day and minute is not permitted.

One or more of millisecond, microsecond and nanosecond can only be specified as long as second is also specified.

Query

```
RETURN localdatetime({year:1984, month:10, day:11, hour:12, minute:31, second:14, millisecond:123, microsecond:456, nanosecond:789}) AS theDate
```

Table 357. Result

<table>
<thead>
<tr>
<th>theDate</th>
</tr>
</thead>
</table>

Rows: 1

Creating a week (Year-Week-Day) LocalDateTime

localdatetime() returns a LocalDateTime value with the specified year, week, dayOfWeek, hour, minute, second, millisecond, microsecond and nanosecond component values.

Syntax: `localdatetime({year [, week, dayOfWeek, hour, minute, second, millisecond, microsecond, nanosecond]})`

Returns:

A LocalDateTime.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>year</td>
<td>An expression consisting of at least four digits that specifies the year.</td>
</tr>
<tr>
<td>week</td>
<td>An integer between 1 and 53 that specifies the week.</td>
</tr>
<tr>
<td>dayOfWeek</td>
<td>An integer between 1 and 7 that specifies the day of the week.</td>
</tr>
<tr>
<td>hour</td>
<td>An integer between 0 and 23 that specifies the hour of the day.</td>
</tr>
<tr>
<td>minute</td>
<td>An integer between 0 and 59 that specifies the number of minutes.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>second</td>
<td>An integer between 0 and 59 that specifies the number of seconds.</td>
</tr>
<tr>
<td>millisecond</td>
<td>An integer between 0 and 999 that specifies the number of milliseconds.</td>
</tr>
<tr>
<td>microsecond</td>
<td>An integer between 0 and 999,999 that specifies the number of microseconds.</td>
</tr>
<tr>
<td>nanosecond</td>
<td>An integer between 0 and 999,999,999 that specifies the number of nanoseconds.</td>
</tr>
</tbody>
</table>

**Considerations:**

The week component will default to 1 if `week` is omitted.

The day of the week component will default to 1 if `dayOfWeek` is omitted.

The hour component will default to 0 if `hour` is omitted.

The minute component will default to 0 if `minute` is omitted.

The second component will default to 0 if `second` is omitted.

Any missing `millisecond`, `microsecond` or `nanosecond` values will default to 0.

If `millisecond`, `microsecond` and `nanosecond` are given in combination (as part of the same set of parameters), the individual values must be in the range 0 to 999.

The least significant components in the set `year`, `week`, `dayOfWeek`, `hour`, `minute`, and `second` may be omitted; i.e. it is possible to specify only `year`, `week` and `dayOfWeek`, but specifying `year`, `week`, `dayOfWeek` and `minute` is not permitted.

One or more of `millisecond`, `microsecond` and `nanosecond` can only be specified as long as `second` is also specified.

**Query**

```sql
RETURN localdatetime({year: 1984, week: 10, dayOfWeek: 3, hour: 12, minute: 31, second: 14, millisecond: 645}) AS theDate
```

**Table 358. Result**

<table>
<thead>
<tr>
<th>theDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984-03-07T12:31:14.645</td>
</tr>
</tbody>
</table>

Rows: 1

**Creating a quarter (Year-Quarter-Day) DateTime**

`localdatetime()` returns a `LocalDateTime` value with the specified `year`, `quarter`, `dayOfWeek`, `hour`, `minute`, `second`, `millisecond`, `microsecond` and `nanosecond` component values.

**Syntax:**

```
localdatetime({year [, quarter, dayOfWeek, hour, minute, second, millisecond, microsecond, nanosecond]})
```

**Returns:**
A LocalDateTime.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>year</td>
<td>An expression consisting of at least four digits that specifies the year.</td>
</tr>
<tr>
<td>quarter</td>
<td>An integer between 1 and 4 that specifies the quarter.</td>
</tr>
<tr>
<td>dayOfQuarter</td>
<td>An integer between 1 and 92 that specifies the day of the quarter.</td>
</tr>
<tr>
<td>hour</td>
<td>An integer between 0 and 23 that specifies the hour of the day.</td>
</tr>
<tr>
<td>minute</td>
<td>An integer between 0 and 59 that specifies the number of minutes.</td>
</tr>
<tr>
<td>second</td>
<td>An integer between 0 and 59 that specifies the number of seconds.</td>
</tr>
<tr>
<td>millisecond</td>
<td>An integer between 0 and 999 that specifies the number of milliseconds.</td>
</tr>
<tr>
<td>microsecond</td>
<td>An integer between 0 and 999,999 that specifies the number of microseconds.</td>
</tr>
<tr>
<td>nanosecond</td>
<td>An integer between 0 and 999,999,999 that specifies the number of nanoseconds.</td>
</tr>
</tbody>
</table>

Considerations:

The quarter component will default to 1 if quarter is omitted.

The day of the quarter component will default to 1 if dayOfQuarter is omitted.

The hour component will default to 0 if hour is omitted.

The minute component will default to 0 if minute is omitted.

The second component will default to 0 if second is omitted.

Any missing millisecond, microsecond or nanosecond values will default to 0.

If millisecond, microsecond and nanosecond are given in combination (as part of the same set of parameters), the individual values must be in the range 0 to 999.

The least significant components in the set year, quarter, dayOfQuarter, hour, minute, and second may be omitted; i.e. it is possible to specify only year, quarter and dayOfQuarter, but specifying year, quarter, dayOfQuarter and minute is not permitted.

One or more of millisecond, microsecond and nanosecond can only be specified as long as second is also specified.
Query

```
RETURN localdatetime({year: 1984, quarter: 3, dayOfQuarter: 45, hour: 12, minute: 31, second: 14, nanosecond: 645876123}) AS theDate
```

Table 359. Result

<table>
<thead>
<tr>
<th>theDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984-08-14T12:31:14.645876123</td>
</tr>
</tbody>
</table>

Rows: 1

Creating an ordinal (Year-Day) LocalDateTime

The `localdatetime()` function returns a `LocalDateTime` value with the specified year, `ordinalDay`, `hour`, `minute`, `second`, `millisecond`, `microsecond` and `nanosecond` component values.

**Syntax:**

```
localdatetime({year [, ordinalDay, hour, minute, second, millisecond, microsecond, nanosecond]})
```

**Returns:**

A `LocalDateTime`.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>year</td>
<td>An expression consisting of at least four digits that specifies the year.</td>
</tr>
<tr>
<td>ordinalDay</td>
<td>An integer between 1 and 366 that specifies the ordinal day of the year.</td>
</tr>
<tr>
<td>hour</td>
<td>An integer between 0 and 23 that specifies the hour of the day.</td>
</tr>
<tr>
<td>minute</td>
<td>An integer between 0 and 59 that specifies the number of minutes.</td>
</tr>
<tr>
<td>second</td>
<td>An integer between 0 and 59 that specifies the number of seconds.</td>
</tr>
<tr>
<td>millisecond</td>
<td>An integer between 0 and 999 that specifies the number of milliseconds.</td>
</tr>
<tr>
<td>microsecond</td>
<td>An integer between 0 and 999,999 that specifies the number of microseconds.</td>
</tr>
<tr>
<td>nanosecond</td>
<td>An integer between 0 and 999,999,999 that specifies the number of nanoseconds.</td>
</tr>
</tbody>
</table>

**Considerations:**
The ordinal day of the year component will default to 1 if `ordinalDay` is omitted.

The hour component will default to 0 if `hour` is omitted.

The minute component will default to 0 if `minute` is omitted.

The second component will default to 0 if `second` is omitted.

Any missing `millisecond`, `microsecond` or `nanosecond` values will default to 0.

If `millisecond`, `microsecond` and `nanosecond` are given in combination (as part of the same set of parameters), the individual values must be in the range 0 to 999.

The least significant components in the set `year`, `ordinalDay`, `hour`, `minute`, and `second` may be omitted; i.e. it is possible to specify only `year` and `ordinalDay`, but specifying `year`, `ordinalDay` and `minute` is not permitted.

One or more of `millisecond`, `microsecond` and `nanosecond` can only be specified as long as `second` is also specified.

Query

```haskell
RETURN localdatetime({year: 1984, ordinalDay: 202, hour: 12, minute: 31, second: 14, microsecond: 645876}) AS theDate
```

Table 360. Result

<table>
<thead>
<tr>
<th>theDate</th>
</tr>
</thead>
</table>

Rows: 1

Creating a `LocalDateTime` from a string

`localdatetime()` returns the `LocalDateTime` value obtained by parsing a string representation of a temporal value.

**Syntax:** `localdatetime(temporalValue)`

**Returns:**

A `LocalDateTime`.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>temporalValue</code></td>
<td>A string representing a temporal value.</td>
</tr>
</tbody>
</table>

**Considerations:**

- `temporalValue` must comply with the format defined for `dates` and `times`.
- `temporalValue` must denote a valid date and time; i.e. a `temporalValue` denoting 30 February 2001 is invalid.
- `localdatetime(null)` returns null.
Query

```sql
UNWIND ['2015-07-21T21:40:32.142',
         '2015-W30-2T214032.142',
         '2015-202T21:40:32',
         '2015202T21']
  AS theDate
RETURN theDate
```

Table 361. Result

<table>
<thead>
<tr>
<th>theDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015-07-21T21:40:32</td>
</tr>
<tr>
<td>2015-07-21T21:40:32</td>
</tr>
<tr>
<td>2015-07-21T21:40</td>
</tr>
<tr>
<td>2015-07-21T21:00</td>
</tr>
</tbody>
</table>

Rows: 4

Creating a `LocalDateTime` using other temporal values as components

`localdatetime()` returns the `LocalDateTime` value obtained by selecting and composing components from another temporal value. In essence, this allows a `Date`, `DateTime`, `Time` or `LocalTime` value to be converted to a `LocalDateTime`, and for "missing" components to be provided.

**Syntax:** `localdatetime({datetime [, year, …, nanosecond]}) | localdatetime({date [, year, …, nanosecond]}) | localdatetime({time [, year, …, nanosecond]}) | localdatetime({date, time [, year, …, nanosecond]})`

**Returns:**

A `LocalDateTime`.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>datetime</td>
<td>A <code>DateTime</code> value.</td>
</tr>
<tr>
<td>date</td>
<td>A <code>Date</code> value.</td>
</tr>
<tr>
<td>time</td>
<td>A <code>Time</code> value.</td>
</tr>
<tr>
<td>year</td>
<td>An expression consisting of at least four digits that specifies the year.</td>
</tr>
<tr>
<td>month</td>
<td>An integer between 1 and 12 that specifies the month.</td>
</tr>
<tr>
<td>day</td>
<td>An integer between 1 and 31 that specifies the day of the month.</td>
</tr>
<tr>
<td>week</td>
<td>An integer between 1 and 53 that specifies the week.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>dayOfWeek</td>
<td>An integer between 1 and 7 that specifies the day of the week.</td>
</tr>
<tr>
<td>quarter</td>
<td>An integer between 1 and 4 that specifies the quarter.</td>
</tr>
<tr>
<td>dayOfQuarter</td>
<td>An integer between 1 and 92 that specifies the day of the quarter.</td>
</tr>
<tr>
<td>ordinalDay</td>
<td>An integer between 1 and 366 that specifies the ordinal day of the year.</td>
</tr>
<tr>
<td>hour</td>
<td>An integer between 0 and 23 that specifies the hour of the day.</td>
</tr>
<tr>
<td>minute</td>
<td>An integer between 0 and 59 that specifies the number of minutes.</td>
</tr>
<tr>
<td>second</td>
<td>An integer between 0 and 59 that specifies the number of seconds.</td>
</tr>
<tr>
<td>millisecond</td>
<td>An integer between 0 and 999 that specifies the number of milliseconds.</td>
</tr>
<tr>
<td>microsecond</td>
<td>An integer between 0 and 999,999 that specifies the number of microseconds.</td>
</tr>
<tr>
<td>nanosecond</td>
<td>An integer between 0 and 999,999,999 that specifies the number of nanoseconds.</td>
</tr>
</tbody>
</table>

Considerations:

If any of the optional parameters are provided, these will override the corresponding components of `datetime`, `date` and/or `time`.

`localdatetime(dd)` may be written instead of `localdatetime({datetime: dd}).`

The following query shows the various usages of `localdatetime({date [, year, …, nanosecond]})`

Query

```sql
WITH dd AS date({year: 1984, month: 10, day: 11})
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 362. Result

<table>
<thead>
<tr>
<th>dateHHMMSS</th>
<th>dateDDHHMMSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984-10-11T10:10</td>
<td>1984-10-28T10:10</td>
</tr>
</tbody>
</table>

Rows: 1

The following query shows the various usages of `localdatetime({time [, year, …, nanosecond]})`
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 364. Result
<table>
<thead>
<tr>
<th>dateTime</th>
<th>dateTimeDDSS</th>
</tr>
</thead>
</table>
Rows: 1
```

The following query shows the various usages of `localdatetime({datetime [, year, …, nanosecond]})`

```
Query
WITH datetime({year: 1984, month: 10, day: 11, hour: 12, timezone: '+01:00'}) as dd
RETURN localdatetime({datetime: dd}) AS dateTime,
     localdatetime({datetime: dd, day: 28, second: 42}) as dateTimeDDSS
Table 365. Result
<table>
<thead>
<tr>
<th>dateTime</th>
<th>dateTimeDDSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984-10-11T12:00</td>
<td>1984-10-28T12:00:42</td>
</tr>
</tbody>
</table>
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 is year in order to ensure the returned value has the day set to x instead of the default day (which is 1).

Syntax: `localdatetime.truncate(unit [, temporalInstantValue [, mapOfComponents ]])`

Returns:

A LocalDateTime.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unit</td>
<td>A string expression evaluating to one of the following: millennium, century, decade, year, weekYear, quarter, month, week, day, hour, minute, second, millisecond, microsecond.</td>
</tr>
<tr>
<td>temporalInstantValue</td>
<td>An expression of one of the following types: {DateTime, LocalDateTime, Date}.</td>
</tr>
<tr>
<td>mapOfComponents</td>
<td>An expression evaluating to a map containing components less significant than unit.</td>
</tr>
</tbody>
</table>

Considerations:

- `temporalInstantValue` cannot be a Date value if `unit` is one of {hour, minute, second, millisecond, microsecond}.
- Any component that is provided in `mapOfComponents` must be less significant than `unit`; i.e. if `unit` is 'day', `mapOfComponents` cannot contain information pertaining to a month.
- Any component that is not contained in `mapOfComponents` and which is less significant than `unit` will be set to its minimal value.
- If `mapOfComponents` is not provided, all components of the returned value which are less significant than `unit` will be set to their default values.
- If `temporalInstantValue` is not provided, it will be set to the current date and time, i.e. `localdatetime.truncate(unit)` is equivalent of `localdatetime.truncate(unit, localdatetime())`.

Query

```sql
WITH localdatetime((year: 2017, month: 11, day: 11, hour: 12, minute: 31, second: 14, nanosecond: 645876123)) AS d
RETURN localdatetime.truncate('millennium', d) AS truncMillenium,
     localdatetime.truncate('year', d, (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 366. Result
10.9.5. LocalTime: `localtime()`

Details for using the `localtime()` function.

- Getting the current LocalTime
- Creating a LocalTime
- Creating a LocalTime from a string
- Creating a LocalTime using other temporal values as components
- Truncating a LocalTime

Getting the current LocalTime

`localtime()` returns the current LocalTime value. If no time zone parameter is specified, the local time zone will be used.

Syntax: `localtime([[timezone]])`

Returns:

A LocalTime.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>timezone</td>
<td>A string expression that represents the time zone</td>
</tr>
</tbody>
</table>

Considerations:

If no parameters are provided, `localtime()` must be invoked (`localtime({})` is invalid).

Query

```sql
RETURN localtime() AS now
```

The current local time (i.e. in the local time zone) is returned.

Table 367. Result
The current local time in California is returned.

Table 368. Result

<table>
<thead>
<tr>
<th>nowInLA</th>
<th>07:22:01.716</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

`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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timezone</td>
<td>A string expression that represents the time zone</td>
</tr>
</tbody>
</table>

**Query**

```
RETURN localtime(transaction()) AS now
```

Table 369. Result

<table>
<thead>
<tr>
<th>now</th>
<th>14:22:01.727</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>
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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timezone</td>
<td>A string expression that represents the time zone</td>
</tr>
</tbody>
</table>

**Query**

```
RETURN localtime.statement() AS now
```

**Table 370. Result**

<table>
<thead>
<tr>
<th>now</th>
</tr>
</thead>
<tbody>
<tr>
<td>14:22:01.741</td>
</tr>
</tbody>
</table>

**Query**

```
RETURN localtime.statement('America/Los Angeles') AS nowInLA
```

**Table 371. Result**

<table>
<thead>
<tr>
<th>nowInLA</th>
</tr>
</thead>
<tbody>
<tr>
<td>07:22:01.756</td>
</tr>
</tbody>
</table>

localtime.realtime()

localtime.realtime() returns the current LocalTime value using the realtime clock. This value will be the live clock of the system.

**Syntax:** `localtime.realtime([[[timezone]]])`

**Returns:**

A LocalTime.
Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timezone</td>
<td>A string expression that represents the timezone</td>
</tr>
</tbody>
</table>

Query

```
RETURN localtime().realtime() AS now
```

Table 372. Result

<table>
<thead>
<tr>
<th>now</th>
</tr>
</thead>
<tbody>
<tr>
<td>14:22:01.796004</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

Creating a LocalTime

`localtime()` returns a LocalTime value with the specified hour, minute, second, millisecond, microsecond and nanosecond component values.

Syntax: `localtime([hour [, minute, second, millisecond, microsecond, nanosecond]])`

Returns:

A LocalTime.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>hour</td>
<td>An integer between 0 and 23 that specifies the hour of the day.</td>
</tr>
<tr>
<td>minute</td>
<td>An integer between 0 and 59 that specifies the number of minutes.</td>
</tr>
<tr>
<td>second</td>
<td>An integer between 0 and 59 that specifies the number of seconds.</td>
</tr>
<tr>
<td>millisecond</td>
<td>An integer between 0 and 999 that specifies the number of milliseconds.</td>
</tr>
<tr>
<td>microsecond</td>
<td>An integer between 0 and 999,999 that specifies the number of microseconds.</td>
</tr>
<tr>
<td>nanosecond</td>
<td>An integer between 0 and 999,999,999 that specifies the number of nanoseconds.</td>
</tr>
</tbody>
</table>

Considerations:

The hour component will default to 0 if hour is omitted.
The minute component will default to 0 if minute is omitted.

The second component will default to 0 if second is omitted.

Any missing millisecond, microsecond or nanosecond values will default to 0.

If millisecond, microsecond and nanosecond are given in combination (as part of the same set of parameters), the individual values must be in the range 0 to 999.

The least significant components in the set hour, minute, and second may be omitted; i.e. it is possible to specify only hour and minute, but specifying hour and second is not permitted.

One or more of millisecond, microsecond and nanosecond can only be specified as long as second is also specified.

---

**Query**

```
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 373. Result**

<table>
<thead>
<tr>
<th>theTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:31:14.123456789</td>
</tr>
<tr>
<td>12:31:14</td>
</tr>
<tr>
<td>12:00</td>
</tr>
</tbody>
</table>

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:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>temporalValue</td>
<td>A string representing a temporal value.</td>
</tr>
</tbody>
</table>

**Considerations:**

- `temporalValue` must comply with the format defined for times.
- `temporalValue` must denote a valid time; i.e. a `temporalValue` denoting 13:46:64 is invalid.
- `localtime(null)` returns null.
UNWIND [localtime('21:40:32.142'),
    localtime('214032.142'),
    localtime('21:40'),
    localtime('21')]
AS theTime
RETURN theTime

Table 374. Result
<table>
<thead>
<tr>
<th>theTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>21:40:32.142</td>
</tr>
<tr>
<td>21:40:32.142</td>
</tr>
<tr>
<td>21:40</td>
</tr>
<tr>
<td>21:00</td>
</tr>
<tr>
<td>Rows: 4</td>
</tr>
</tbody>
</table>

Creating a LocalTime using other temporal values as components

`localtime()` returns the LocalTime value obtained by selecting and composing components from another temporal value. In essence, this allows a DateTime, LocalDateTime or Time value to be converted to a LocalTime, and for "missing" components to be provided.

**Syntax:** `localtime({time [, hour, ..., nanosecond]})`

**Returns:**

A LocalTime.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>time</td>
<td>A Time value.</td>
</tr>
<tr>
<td>hour</td>
<td>An integer between 0 and 23 that specifies the hour of the day.</td>
</tr>
<tr>
<td>minute</td>
<td>An integer between 0 and 59 that specifies the number of minutes.</td>
</tr>
<tr>
<td>second</td>
<td>An integer between 0 and 59 that specifies the number of seconds.</td>
</tr>
<tr>
<td>millisecond</td>
<td>An integer between 0 and 999 that specifies the number of milliseconds.</td>
</tr>
<tr>
<td>microsecond</td>
<td>An integer between 0 and 999,999 that specifies the number of microseconds.</td>
</tr>
</tbody>
</table>
Considerations:

If any of the optional parameters are provided, these will override the corresponding components of `time`.

`localtime(tt)` may be written instead of `localtime({time: tt})`.

Query

```
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 375. Result

<table>
<thead>
<tr>
<th>timeOnly</th>
<th>timeSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:31:14.645876</td>
<td>12:31:42.645876</td>
</tr>
<tr>
<td>Rows: 1</td>
<td></td>
</tr>
</tbody>
</table>

Truncating a `LocalTime`

`localtime.truncate()` returns the `LocalTime` value obtained by truncating a specified temporal instant value at the nearest preceding point in time at the specified component boundary (which is denoted by the truncation unit passed as a parameter to the function). In other words, the `LocalTime` returned will have all components that are less significant than the specified truncation unit set to their default values.

It is possible to supplement the truncated value by providing a map containing components which are less significant than the truncation unit. This will have the effect of overriding the default values which would otherwise have been set for these less significant components. For example, `minute` — with some value `x` — may be provided when the truncation unit is `hour` in order to ensure the returned value has the minute set to `x` instead of the default minute (which is 1).

Syntax: `localtime.truncate(unit [, temporalInstantValue [, mapOfComponents ] ])

Returns:

A `LocalTime`.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unit</td>
<td>A string expression evaluating to one of the following: [day, hour, minute, second, millisecond, microsecond].</td>
</tr>
<tr>
<td>temporalInstantValue</td>
<td>An expression of one of the following types: [DateTime, LocalDateTime, Time, LocalTime].</td>
</tr>
</tbody>
</table>
### Name

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mapOfComponents An expression evaluating to a map containing components less significant than unit.</td>
</tr>
</tbody>
</table>

#### Considerations:

<table>
<thead>
<tr>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any component that is provided in mapOfComponents must be less significant than unit; i.e. if unit is 'second', mapOfComponents cannot contain information pertaining to a minute.</td>
</tr>
<tr>
<td>Any component that is not contained in mapOfComponents and which is less significant than unit will be set to its minimal value.</td>
</tr>
<tr>
<td>If mapOfComponents is not provided, all components of the returned value which are less significant than unit will be set to their default values.</td>
</tr>
<tr>
<td>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()).</td>
</tr>
</tbody>
</table>

#### Query

```sql
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 376. Result

<table>
<thead>
<tr>
<th>truncDay</th>
<th>truncHour</th>
<th>truncMinute</th>
<th>truncSecond</th>
<th>truncMillisecond</th>
<th>truncMicrosecond</th>
</tr>
</thead>
</table>

Rows: 1

### 10.9.6. Time: time()

Details for using the time() function.

- Getting the current Time
- Creating a Time
- Creating a Time from a string
- Creating a Time using other temporal values as components
- Truncating a Time

**Getting the current Time**

time() returns the current Time value. If no time zone parameter is specified, the local time zone will be
used.

Syntax: `time([timezone])`

Returns:

A Time.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>timezone</td>
<td>A string expression that represents the time zone</td>
</tr>
</tbody>
</table>

Considerations:

If no parameters are provided, `time()` must be invoked (`time({})` is invalid).

Query

```
RETURN time() AS currentTime
```

The current time of day using the local time zone is returned.

Table 377. Result

<table>
<thead>
<tr>
<th>currentTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>14:22:01.918Z</td>
</tr>
</tbody>
</table>

Rows: 1

Query

```
RETURN time({timezone: 'America/Los Angeles'}) AS currentTimeInLA
```

The current time of day in California is returned.

Table 378. Result

<table>
<thead>
<tr>
<th>currentTimeInLA</th>
</tr>
</thead>
<tbody>
<tr>
<td>07:22:01.929-07:00</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timezone</td>
<td>A string expression that represents the timezone</td>
</tr>
</tbody>
</table>

Query

```
RETURN time.transaction() AS currentTime
```

Table 379. Result

<table>
<thead>
<tr>
<th>currentTime</th>
<th>14:22:01.940Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows:</td>
<td>1</td>
</tr>
</tbody>
</table>

`time.statement()`

`time.statement()` returns the current Time value using the statement clock. This value will be the same for each invocation within the same statement. However, a different value may be produced for different statements within the same transaction.

Syntax: `time.statement([timezone])`

Returns:

A Time.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timezone</td>
<td>A string expression that represents the timezone</td>
</tr>
</tbody>
</table>

Query

```
RETURN time.statement() AS currentTime
```

Table 380. Result

<table>
<thead>
<tr>
<th>currentTime</th>
<th>14:22:01.950Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows:</td>
<td>1</td>
</tr>
</tbody>
</table>
Query

```
RETURN time.statement('America/Los Angeles') AS currentTimeInLA
```

Table 381. Result

<table>
<thead>
<tr>
<th>currentTimeInLA</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>07:22:01.960-07:00</td>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

```
time.realtime()
```

`time.realtime()` returns the current Time value using the `realtime` clock. This value will be the live clock of the system.

**Syntax:** `time.realtime([[timezone]])`

**Returns:**

A Time.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timezone</td>
<td>A string expression that represents the timezone</td>
</tr>
</tbody>
</table>

Query

```
RETURN time.realtime() AS currentTime
```

Table 382. Result

<table>
<thead>
<tr>
<th>currentTime</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>14:22:01.982244Z</td>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

Creating a Time

`time()` returns a Time value with the specified hour, minute, second, millisecond, microsecond, nanosecond and timezone component values.

**Syntax:** `time([hour [, minute, second, millisecond, microsecond, nanosecond, timezone]])`

**Returns:**

A Time.
Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>hour</td>
<td>An integer between 0 and 23 that specifies the hour of the day.</td>
</tr>
<tr>
<td>minute</td>
<td>An integer between 0 and 59 that specifies the number of minutes.</td>
</tr>
<tr>
<td>second</td>
<td>An integer between 0 and 59 that specifies the number of seconds.</td>
</tr>
<tr>
<td>millisecond</td>
<td>An integer between 0 and 999 that specifies the number of milliseconds.</td>
</tr>
<tr>
<td>microsecond</td>
<td>An integer between 0 and 999,999 that specifies the number of microseconds.</td>
</tr>
<tr>
<td>nanosecond</td>
<td>An integer between 0 and 999,999,999 that specifies the number of nanoseconds.</td>
</tr>
<tr>
<td>timezone</td>
<td>An expression that specifies the time zone.</td>
</tr>
</tbody>
</table>

Considerations:

- The hour component will default to 0 if hour is omitted.
- The minute component will default to 0 if minute is omitted.
- The second component will default to 0 if second is omitted.
- Any missing millisecond, microsecond or nanosecond values will default to 0.
- The timezone component will default to the configured default time zone if timezone is omitted.
- If millisecond, microsecond and nanosecond are given in combination (as part of the same set of parameters), the individual values must be in the range 0 to 999.
- The least significant components in the set hour, minute, and second may be omitted; i.e. it is possible to specify only hour and minute, but specifying hour and second is not permitted.
- One or more of millisecond, microsecond and nanosecond can only be specified as long as second is also specified.

Query

```sql
UNWIND [
  time({hour: 12, minute: 31, second: 14, millisecond: 123, microsecond: 456, nanosecond: 789}),
  time({hour: 12, minute: 31, second: 14, nanosecond: 645876123}),
  time({hour: 12, minute: 31, second: 14, microsecond: 645876, timezone: '+01:00'}),
  time({hour: 12, minute: 31, timezone: '+01:00'})
] AS theTime
RETURN theTime
```

Table 383. Result

<table>
<thead>
<tr>
<th>theTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:31:14.123456789Z</td>
</tr>
</tbody>
</table>
Creating a Time from a string

time() returns the Time value obtained by parsing a string representation of a temporal value.

Syntax: time(temporalValue)

Returns:
A Time.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>temporalValue</td>
<td>A string representing a temporal value.</td>
</tr>
</tbody>
</table>

Considerations:

temporalValue must comply with the format defined for times and time zones.
The timezone component will default to the configured default time zone if it is omitted.
temporalValue must denote a valid time; i.e. a temporalValue denoting 15:67 is invalid.
time(null) returns null.

Query

```sql
UNWIND [
    time('21:40:32.142+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 384. Result

<table>
<thead>
<tr>
<th>theTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>21:40:32.142+01:00</td>
</tr>
<tr>
<td>21:40:32.142Z</td>
</tr>
</tbody>
</table>
Creating a Time using other temporal values as components

`time()` returns the Time value obtained by selecting and composing components from another temporal value. In essence, this allows a `DateTime`, `LocalDateTime` or `LocalTime` value to be converted to a `Time`, and for "missing" components to be provided.

**Syntax:** `time({time [, hour, ..., timezone]})`

**Returns:**
A Time.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>time</code></td>
<td>A Time value.</td>
</tr>
<tr>
<td><code>hour</code></td>
<td>An integer between 0 and 23 that specifies the hour of the day.</td>
</tr>
<tr>
<td><code>minute</code></td>
<td>An integer between 0 and 59 that specifies the number of minutes.</td>
</tr>
<tr>
<td><code>second</code></td>
<td>An integer between 0 and 59 that specifies the number of seconds.</td>
</tr>
<tr>
<td><code>millisecond</code></td>
<td>An integer between 0 and 999 that specifies the number of milliseconds.</td>
</tr>
<tr>
<td><code>microsecond</code></td>
<td>An integer between 0 and 999,999 that specifies the number of microseconds.</td>
</tr>
<tr>
<td><code>nanosecond</code></td>
<td>An integer between 0 and 999,999,999 that specifies the number of nanoseconds.</td>
</tr>
<tr>
<td><code>timezone</code></td>
<td>An expression that specifies the time zone.</td>
</tr>
</tbody>
</table>

**Considerations:**
If any of the optional parameters are provided, these will override the corresponding components of `time`. 
time(tt) may be written instead of time({time: tt}).

Selecting a Time or DateTime value as the time component also selects its time zone. If a LocalTime or LocalDateTime is selected instead, the default time zone is used. In any case, the time zone can be overridden explicitly.

Selecting a DateTime or Time as the time component and overwriting the time zone will adjust the local time to keep the same point in time.

Query

```sql
WITH localtime({hour: 12, minute: 31, second: 14, microsecond: 645876}) AS tt
RETURN
  time({time: tt}) AS timeOnly,
  time({time: tt, timezone: '+05:00'}) AS timezone,
  time({time: tt, second: 42}) AS timeSS,
  time({time: tt, second: 42, timezone: '+05:00'}) AS timeSSTimezone
```

Table 385. Result

<table>
<thead>
<tr>
<th>timeOnly</th>
<th>timeZone</th>
<th>timeSS</th>
<th>timeSSTimezone</th>
</tr>
</thead>
</table>

Rows: 1

Truncating a Time

time.truncate() returns the Time value obtained by truncating a specified temporal instant value at the nearest preceding point in time at the specified component boundary (which is denoted by the truncation unit passed as a parameter to the function). In other words, the Time returned will have all components that are less significant than the specified truncation unit set to their default values.

It is possible to supplement the truncated value by providing a map containing components which are less significant than the truncation unit. This will have the effect of overriding the default values which would otherwise have been set for these less significant components. For example, minute — with some value x — may be provided when the truncation unit is hour in order to ensure the returned value has the minute set to x instead of the default minute (which is 1).

Syntax: `time.truncate(unit [, temporalInstantValue [, mapOfComponents ]])`

Returns:

A Time.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unit</td>
<td>A string expression evaluating to one of the following: {day, hour, minute, second, millisecond, microsecond}.</td>
</tr>
<tr>
<td>temporalInstantValue</td>
<td>An expression of one of the following types: {DateTime, LocalDateTime, Time, LocalTime}.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>mapOfComponents</td>
<td>An expression evaluating to a map containing components less significant than unit. During truncation, a time zone can be attached or overridden using the key timezone.</td>
</tr>
</tbody>
</table>

**Considerations:**

Truncating time to day — i.e. unit is 'day' — is supported, and yields midnight at the start of the day (00:00), regardless of the value of temporalInstantValue. However, the time zone of temporalInstantValue is retained.

The time zone of temporalInstantValue may be overridden; for example, `time.truncate('minute', input, {timezone: '+0200'})`.

If `temporalInstantValue` is one of {Time, DateTime} — a value with a time zone — and the time zone is overridden, no time conversion occurs.

If `temporalInstantValue` is one of {LocalTime, LocalDateTime, Date} — a value without a time zone — and the time zone is not overridden, the configured default time zone will be used.

Any component that is provided in mapOfComponents must be less significant than unit; i.e. if unit is 'second', mapOfComponents cannot contain information pertaining to a minute.

Any component that is not contained in mapOfComponents and which is less significant than unit will be set to its minimal value.

If `mapOfComponents` is not provided, all components of the returned value which are less significant than unit will be set to their default values.

If `temporalInstantValue` is not provided, it will be set to the current time and timezone, i.e. `time.truncate(unit)` is equivalent of `time.truncate(unit, time())`.

**Query**

```cypher
WITH time({hour: 12, minute: 31, second: 14, nanosecond: 645876123, timezone: '-01:00'}) AS t
RETURN time.truncate('day', t) AS truncDay,
    time.truncate('hour', t) AS truncHour,
    time.truncate('minute', t) AS truncMinute,
    time.truncate('second', t) AS truncSecond,
    time.truncate('millisecond', t, {nanosecond: 2}) AS truncMillisecond,
    time.truncate('microsecond', t) AS truncMicrosecond
```

**Table 386. Result**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>truncDay</td>
<td>truncHour</td>
<td>truncMinute</td>
<td>truncSecond</td>
<td>truncMillisecond</td>
</tr>
<tr>
<td>00:00-01:00</td>
<td>12:00-01:00</td>
<td>12:31-01:00</td>
<td>12:31:14-01:00</td>
<td>12:31:14.6450000000</td>
</tr>
</tbody>
</table>

Rows: 1

**10.10. Temporal functions - duration**

`Cypher` provides functions allowing for the creation and manipulation of values for a Duration temporal type.
duration():

- Creating a Duration from duration components
- Creating a Duration from a string
- Computing the Duration between two temporal instants

Information regarding specifying and accessing components of a Duration value can be found [here](#).

10.10.1. Creating a Duration from duration components

duration() can construct a Duration from a map of its components in the same way as the temporal instant types.

- years
- quarters
- months
- weeks
- days
- hours
- minutes
- seconds
- milliseconds
- microseconds
- nanoseconds

Syntax: `duration([ {years, quarters, months, weeks, days, hours, minutes, seconds, milliseconds, microseconds, nanoseconds} ])`

Returns:

A Duration.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>years</td>
<td>A numeric expression.</td>
</tr>
<tr>
<td>quarters</td>
<td>A numeric expression.</td>
</tr>
<tr>
<td>months</td>
<td>A numeric expression.</td>
</tr>
<tr>
<td>weeks</td>
<td>A numeric expression.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>days</td>
<td>A numeric expression.</td>
</tr>
<tr>
<td>hours</td>
<td>A numeric expression.</td>
</tr>
<tr>
<td>minutes</td>
<td>A numeric expression.</td>
</tr>
<tr>
<td>seconds</td>
<td>A numeric expression.</td>
</tr>
<tr>
<td>milliseconds</td>
<td>A numeric expression.</td>
</tr>
<tr>
<td>microseconds</td>
<td>A numeric expression.</td>
</tr>
<tr>
<td>nanoseconds</td>
<td>A numeric expression.</td>
</tr>
</tbody>
</table>

**Considerations:**

At least one parameter must be provided (duration() and duration({}) are invalid).

There is no constraint on how many of the parameters are provided.

It is possible to have a Duration where the amount of a smaller unit (e.g. seconds) exceeds the threshold of a larger unit (e.g. days).

The values of the parameters may be expressed as decimal fractions.

The values of the parameters may be arbitrarily large.

The values of the parameters may be negative.

**Query**

```graphql
UNWIND [
  duration({days: 14, hours: 16, minutes: 12}),
  duration({months: 5, days: 1.5}),
  duration({months: 0.75}),
  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 387. Result**

<table>
<thead>
<tr>
<th>aDuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>P14DT16H12M</td>
</tr>
<tr>
<td>PSM1DT12H</td>
</tr>
<tr>
<td>P22DT19H51M49.5S</td>
</tr>
<tr>
<td>P17DT12H</td>
</tr>
<tr>
<td>PT1M31.123456789S</td>
</tr>
<tr>
<td>PT1M31.123456789S</td>
</tr>
</tbody>
</table>

Rows: 6
10.10.2. Creating a Duration from a string

duration() returns the Duration value obtained by parsing a string representation of a temporal amount.

Syntax: duration(temporalAmount)

Returns:

A Duration.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>temporalAmount</td>
<td>A string representing a temporal amount.</td>
</tr>
</tbody>
</table>

Considerations:

temporalAmount must comply with either the unit based form or date-and-time based form defined for Durations.

Query

```
UNWIND [
    duration("P14DT16H12M"),
    duration("P9M1.5D"),
    duration("P0.75M"),
    duration("PT0.75M"),
    duration("P2012-02-02T14:37:21.545")
] AS aDuration
RETURN aDuration
```

Table 388. Result

<table>
<thead>
<tr>
<th>aDuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>P14DT16H12M</td>
</tr>
<tr>
<td>P9M1.5D</td>
</tr>
<tr>
<td>P22DT19H51M49.5S</td>
</tr>
<tr>
<td>PT45S</td>
</tr>
<tr>
<td>P2012Y2M2DT14H37M21.545</td>
</tr>
</tbody>
</table>

Rows: 5

10.10.3. Computing the Duration between two temporal instants

duration() has sub-functions which compute the logical difference (in days, months, etc) between two temporal instant values:

- duration.between(a, b): Computes the difference in multiple components between instant a and instant b. This captures month, days, seconds and sub-seconds differences separately.
- duration.inMonths(a, b): Computes the difference in whole months (or quarters or years) between instant a and instant b. This captures the difference as the total number of months. Any difference
smaller than a whole month is disregarded.

- **duration.inDays(a, b)**: Computes the difference in whole days (or weeks) between instant `a` and instant `b`. This captures the difference as the total number of days. Any difference smaller than a whole day is disregarded.

- **duration.inSeconds(a, b)**: Computes the difference in seconds (and fractions of seconds, or minutes or hours) between instant `a` and instant `b`. This captures the difference as the total number of seconds.

duration.between()

duration.between() returns the Duration value equal to the difference between the two given instants.

**Syntax:** `duration.between(instant\_1, instant\_2)`

**Returns:** A Duration.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>instant\_1</code></td>
<td>An expression returning any temporal instant type (Date etc) that represents the starting instant.</td>
</tr>
<tr>
<td><code>instant\_2</code></td>
<td>An expression returning any temporal instant type (Date etc) that represents the ending instant.</td>
</tr>
</tbody>
</table>

**Considerations:**

- If `instant\_1` occurs earlier than `instant\_2`, the resulting Duration will be negative.
- If `instant\_1` has a time component and `instant\_2` does not, the time component of `instant\_1` is assumed to be midnight, and vice versa.
- If `instant\_1` has a time zone component and `instant\_2` does not, the time zone component of `instant\_1` is assumed to be the same as that of `instant\_2`, and vice versa.
- If `instant\_1` has a date component and `instant\_2` does not, the date component of `instant\_1` is assumed to be the same as that of `instant\_2`, and vice versa.

**Query**

```sql
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(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 389. Result**
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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>instant1</td>
<td>An expression returning any temporal instant type (Date etc) that represents the starting instant.</td>
</tr>
<tr>
<td>instant2</td>
<td>An expression returning any temporal instant type (Date etc) that represents the ending instant.</td>
</tr>
</tbody>
</table>

Considerations:

If instant1 occurs earlier than instant2, the resulting Duration will be negative.

If instant1 has a time component and instant2 does not, the time component of instant1 is assumed to be midnight, and vice versa.

If instant1 has a time zone component and instant2 does not, the time zone component of instant2 is assumed to be the same as that of instant1, and vice versa.

If instant1 has a date component and instant2 does not, the date component of instant2 is assumed to be the same as that of instant1, and vice versa.

Any difference smaller than a whole month is disregarded.
UNWIND [  
  duration.inMonths(date("1984-10-11"), date("1985-11-25")),  
  duration.inMonths(date("1985-11-25"), date("1984-10-11")),  
  duration.inMonths(date("1984-10-11"), datetime("1984-10-12T21:40:32.142+0100")),  
  duration.inMonths(date("2015-06-24"), localtime("14:30")),  
  duration.inMonths(datetime({year: 2017, month: 10, day: 29, hour: 0, timezone: 'Europe/Stockholm'}),  
  datetime({year: 2017, month: 10, day: 29, hour: 0, timezone: 'Europe/London'}))  
] AS aDuration  
RETURN aDuration

<table>
<thead>
<tr>
<th>Table 390. Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>aDuration</td>
</tr>
<tr>
<td>P1Y1M</td>
</tr>
<tr>
<td>P-1Y-1M</td>
</tr>
<tr>
<td>PT0S</td>
</tr>
<tr>
<td>PT0S</td>
</tr>
<tr>
<td>PT0S</td>
</tr>
<tr>
<td>P1Y</td>
</tr>
<tr>
<td>Row: 6</td>
</tr>
</tbody>
</table>

duration.inDays()

duration.inDays() returns the Duration value equal to the difference in whole days or weeks between the two given instants.

Syntax: \texttt{duration.inDays(instant\(_1\), instant\(_2\))}

Returns:

A Duration.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>instant(_1)</td>
<td>An expression returning any temporal instant type (Date etc) that represents the starting instant.</td>
</tr>
<tr>
<td>instant(_2)</td>
<td>An expression returning any temporal instant type (Date etc) that represents the ending instant.</td>
</tr>
</tbody>
</table>

Considerations:

- If \texttt{instant\(_1\)} occurs earlier than \texttt{instant\(_2\)}, the resulting Duration will be negative.
- If \texttt{instant\(_1\)} has a time component and \texttt{instant\(_2\)} does not, the time component of \texttt{instant\(_1\)} is assumed to be midnight, and vice versa.
If \( \text{instant}_1 \) has a time zone component and \( \text{instant}_2 \) does not, the time zone component of \( \text{instant}_2 \) is assumed to be the same as that of \( \text{instant}_1 \), and vice versa.

If \( \text{instant}_1 \) has a date component and \( \text{instant}_2 \) does not, the date component of \( \text{instant}_2 \) is assumed to be the same as that of \( \text{instant}_1 \), and vice versa.

Any difference smaller than a whole day is disregarded.

**Query**

```
UNWIND [
    duration.inDays(date("1984-10-11"), date("1985-11-25")),
    duration.inDays(date("1985-11-25"), date("1984-10-11")),
    duration.inDays(date("1985-11-25"), datetime("1984-10-12T21:40:32.142+0100")),
    duration.inDays(date("1984-10-11"), localtime("1984-10-12T00:00:00")),
    duration.inDays(localdatetime("2015-06-24"), localtime("2015-06-24")),
    duration.inDays(date("2015-07-21T21:40:32.142"), localtime("2016-07-21T21:45:22.142")),
    duration.inDays(datetime({year: 2015, month: 10, day: 29}), datetime({year: 2015, month: 10, day: 29, hour: 0, timezone: 'Europe/Stockholm'})),
    duration.inDays(datetime({year: 2015, month: 10, day: 29}), datetime({year: 2015, month: 10, day: 29, hour: 0, timezone: 'Europe/London'}))
] AS aDuration
RETURN aDuration
```

**Table 391. Result**

<table>
<thead>
<tr>
<th>aDuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>P410D</td>
</tr>
<tr>
<td>P-410D</td>
</tr>
<tr>
<td>P1D</td>
</tr>
<tr>
<td>PT0S</td>
</tr>
<tr>
<td>P366D</td>
</tr>
<tr>
<td>PT0S</td>
</tr>
</tbody>
</table>

Rows: 6

**duration.inSeconds()**

duration.inSeconds() returns the Duration value equal to the difference in seconds and fractions of seconds, or minutes or hours, between the two given instants.

**Syntax:** `duration.inSeconds(instant1, instant2)`

**Returns:**

A Duration.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>instant1</td>
<td>An expression returning any temporal instant type (Date etc) that represents the starting instant.</td>
</tr>
<tr>
<td>instant2</td>
<td>An expression returning any temporal instant type (Date etc) that represents the ending instant.</td>
</tr>
</tbody>
</table>
Considerations:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>If <code>instant_1</code> occurs earlier than <code>instant_2</code>, the resulting Duration will be negative.</td>
<td></td>
</tr>
<tr>
<td>If <code>instant_1</code> has a time component and <code>instant_2</code> does not, the time component of <code>instant_1</code> is assumed to be midnight, and vice versa.</td>
<td></td>
</tr>
<tr>
<td>If <code>instant_1</code> has a time zone component and <code>instant_2</code> does not, the time zone component of <code>instant_1</code> is assumed to be the same as that of <code>instant_2</code>, and vice versa.</td>
<td></td>
</tr>
<tr>
<td>If <code>instant_1</code> has a date component and <code>instant_2</code> does not, the date component of <code>instant_1</code> is assumed to be the same as that of <code>instant_2</code>, and vice versa.</td>
<td></td>
</tr>
</tbody>
</table>

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 392. Result

<table>
<thead>
<tr>
<th>aDuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>PT24H</td>
</tr>
<tr>
<td>PT-24H</td>
</tr>
<tr>
<td>PT25H32.142S</td>
</tr>
<tr>
<td>PT1H30M</td>
</tr>
<tr>
<td>PT1H</td>
</tr>
</tbody>
</table>

Rows: 5

10.11. Spatial functions

These functions are used to specify 2D or 3D points in a Coordinate Reference System (CRS) and to calculate the geodesic distance between two points.

Functions:

- `distance()`
- `point()` - WGS 84 2D
- `point()` - WGS 84 3D
- `point()` - Cartesian 2D
- `point()` - Cartesian 3D

The following graph is used for some of the examples below.
10.11.1. distance()

distance() returns a floating point number representing the geodesic distance between two points in the same Coordinate Reference System (CRS).

- If the points are in the Cartesian CRS (2D or 3D), then the units of the returned distance will be the same as the units of the points, calculated using Pythagoras' theorem.
- If the points are in the WGS-84 CRS (2D), then the units of the returned distance will be meters, based on the haversine formula over a spherical earth approximation.
- If the points are in the WGS-84 CRS (3D), then the units of the returned distance will be meters.
  - The distance is calculated in two steps.
    - First, a haversine formula over a spherical earth is used, at the average height of the two points.
    - To account for the difference in height, Pythagoras' theorem is used, combining the previously calculated spherical distance with the height difference.
  - This formula works well for points close to the earth’s surface; for instance, it is well-suited for calculating the distance of an airplane flight. It is less suitable for greater heights, however, such as when calculating the distance between two satellites.

Syntax: distance(point1, point2)

Returns:

A Float.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>point1</td>
<td>A point in either a geographic or cartesian coordinate system.</td>
</tr>
<tr>
<td>point2</td>
<td>A point in the same CRS as 'point1'.</td>
</tr>
</tbody>
</table>

Considerations:

distance(null, null), distance(null, point2) and distance(point1, null) all return null.
Attempting to use points with different Coordinate Reference Systems (such as WGS 84 2D and WGS 84 3D) will return `null`.

Query

```sql
WITH point({x: 2.3, y: 4.5, crs: 'cartesian'}) AS p1, point({x: 1.1, y: 5.4, crs: 'cartesian'}) AS p2
RETURN distance(p1, p2) AS dist
```

The distance between two 2D points in the Cartesian CRS is returned.

Table 393. Result

<table>
<thead>
<tr>
<th>dist</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.5</td>
</tr>
</tbody>
</table>

Rows: 1

Query

```sql
WITH point({longitude: 12.78, latitude: 56.7, height: 100}) as p1,
点({latitude: 56.71, longitude: 12.79, height: 100}) as p2
RETURN distance(p1, p2) as dist
```

The distance between two 3D points in the WGS 84 CRS is returned.

Table 394. Result

<table>
<thead>
<tr>
<th>dist</th>
</tr>
</thead>
<tbody>
<tr>
<td>1269.9148706779097</td>
</tr>
</tbody>
</table>

Rows: 1

Query

```sql
MATCH (t:TrainStation)-[:TRAVEL_ROUTE]->(o:Office)
WITH point({longitude: t.longitude, latitude: t.latitude}) AS trainPoint,
点({longitude: o.longitude, latitude: o.latitude}) AS officePoint
RETURN round(distance(trainPoint, officePoint)) AS travelDistance
```

The distance between the train station in Copenhagen and the Neo4j office in Malmo is returned.

Table 395. Result

<table>
<thead>
<tr>
<th>travelDistance</th>
</tr>
</thead>
<tbody>
<tr>
<td>27842.0</td>
</tr>
</tbody>
</table>

Rows: 1

Query

```sql
RETURN distance(null, point({longitude: 56.7, latitude: 12.78})) AS d
```

If `null` is provided as one or both of the arguments, `null` is returned.
10.11.2. point() - WGS 84 2D

`point({longitude | x, latitude | y [, crs][, srid]})` returns a 2D point in the WGS 84 CRS corresponding to the given coordinate values.

**Syntax:** `point({longitude | x, latitude | y [, crs][, srid]})`

**Returns:**
A 2D point in WGS 84.

**Arguments:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single map consisting of the following:</td>
<td></td>
</tr>
<tr>
<td>longitude/x</td>
<td>A numeric expression that represents the longitude/x value in decimal degrees</td>
</tr>
<tr>
<td>latitude/y</td>
<td>A numeric expression that represents the latitude/y value in decimal degrees</td>
</tr>
<tr>
<td>crs</td>
<td>The optional string 'WGS-84'</td>
</tr>
<tr>
<td>srid</td>
<td>The optional number 4326</td>
</tr>
</tbody>
</table>

**Considerations:**

If any argument provided to `point()` is `null`, `null` will be returned.

If the coordinates are specified using `latitude` and `longitude`, the `crs` or `srid` fields are optional and inferred to be 'WGS-84' (srid=4326).

If the coordinates are specified using `x` and `y`, then either the `crs` or `srid` field is required if a geographic CRS is desired.

**Query**

```
RETURN point({longitude: 56.7, latitude: 12.78}) AS point
```

A 2D point with a `longitude` of 56.7 and a `latitude` of 12.78 in the WGS 84 CRS is returned.
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 398. Result

<table>
<thead>
<tr>
<th>point</th>
</tr>
</thead>
<tbody>
<tr>
<td>point({x: 2.3, y: 4.5, crs: 'wgs-84'})</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

Query

```
MATCH (p:Office)
RETURN point({longitude: p.longitude, latitude: p.latitude}) AS officePoint
```

A 2D point representing the coordinates of the city of Malmo in the WGS 84 CRS is returned.

Table 399. Result

<table>
<thead>
<tr>
<th>officePoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>point({x: 12.994341, y: 55.611784, crs: 'wgs-84'})</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

Query

```
RETURN point(null) AS p
```

If `null` is provided as the argument, `null` is returned.

Table 400. Result

<table>
<thead>
<tr>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;null&gt;</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

### 10.11.3. `point()` - WGS 84 3D

`point({longitude | x, latitude | y, height | z, [, crs][, srid]})` returns a 3D point in the WGS 84 CRS corresponding to the given coordinate values.

**Syntax:** `point({longitude | x, latitude | y, height | z, [, crs][, srid]})`

**Returns:**

A 3D point in WGS 84.
Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>longitude/x</td>
<td>A numeric expression that represents the longitude/x value in decimal degrees</td>
</tr>
<tr>
<td>latitude/y</td>
<td>A numeric expression that represents the latitude/y value in decimal degrees</td>
</tr>
<tr>
<td>height/z</td>
<td>A numeric expression that represents the height/z value in meters</td>
</tr>
<tr>
<td>crs</td>
<td>The optional string 'WGS-84-3D'</td>
</tr>
<tr>
<td>srid</td>
<td>The optional number 4979</td>
</tr>
</tbody>
</table>

Considerations:

If any argument provided to `point()` is `null`, `null` will be returned.

If the `height/z` key and value is not provided, a 2D point in the WGS 84 CRS will be returned.

If the coordinates are specified using `latitude` and `longitude`, the `crs` or `srid` fields are optional and inferred to be 'WGS-84-3D' (srid=4979).

If the coordinates are specified using `x` and `y`, then either the `crs` or `srid` field is required if a geographic CRS is desired.

Query

```
RETURN point({longitude: 56.7, latitude: 12.78, height: 8}) AS point
```

A 3D point with a `longitude` of 56.7, a `latitude` of 12.78 and a height of 8 meters in the WGS 84 CRS is returned.

Table 401. Result

<table>
<thead>
<tr>
<th>point</th>
</tr>
</thead>
<tbody>
<tr>
<td>point({x: 56.7, y: 12.78, z: 8.0, crs: 'wgs-84-3d'})</td>
</tr>
</tbody>
</table>

Rows: 1

10.11.4. `point()` - Cartesian 2D

`point({x, y [, crs][, srid]})` returns a 2D point in the Cartesian CRS corresponding to the given coordinate values.

Syntax: `point({x, y [, crs][, srid]})`

Returns:

A 2D point in Cartesian.
Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A single map consisting of the following:</strong></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>A numeric expression</td>
</tr>
<tr>
<td>y</td>
<td>A numeric expression</td>
</tr>
<tr>
<td>crs</td>
<td>The optional string 'cartesian'</td>
</tr>
<tr>
<td>srid</td>
<td>The optional number 7203</td>
</tr>
</tbody>
</table>

Considerations:

If any argument provided to `point()` is `null`, `null` will be returned.

The `crs` or `srid` fields are optional and default to the Cartesian CRS (which means `srid:7203`).

Query

```
RETURN point({x: 2.3, y: 4.5}) AS point
```

A 2D point with an `x` coordinate of `2.3` and a `y` coordinate of `4.5` in the Cartesian CRS is returned.

Table 402. Result

<table>
<thead>
<tr>
<th>point</th>
</tr>
</thead>
<tbody>
<tr>
<td>point({x: 2.3, y: 4.5, crs: 'cartesian'})</td>
</tr>
<tr>
<td>Rows: 1</td>
</tr>
</tbody>
</table>

10.11.5. `point()` - Cartesian 3D

`point({x, y, z, [, crs][, srid]})` returns a 3D point in the Cartesian CRS corresponding to the given coordinate values.

Syntax: `point({x, y, z, [, crs][, srid]})`

Returns:

A 3D point in Cartesian.

Arguments:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A single map consisting of the following:</strong></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>A numeric expression</td>
</tr>
<tr>
<td>y</td>
<td>A numeric expression</td>
</tr>
<tr>
<td>z</td>
<td>A numeric expression</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>crs</td>
<td>The optional string 'cartesian-3D'</td>
</tr>
<tr>
<td>srid</td>
<td>The optional number 9157</td>
</tr>
</tbody>
</table>

**Considerations:**

If any argument provided to `point()` is `null`, `null` will be returned.

If the `z` key and value is not provided, a 2D point in the Cartesian CRS will be returned.

The `crs` or `srid` fields are optional and default to the 3D Cartesian CRS (which means `srid:9157`).

**Query**

```
RETURN point({x: 2.3, y: 4.5, z: 2}) AS point
```

A 3D point with an `x` coordinate of `2.3`, a `y` coordinate of `4.5` and a `z` coordinate of `2` in the Cartesian CRS is returned.

**Table 403. Result**

<table>
<thead>
<tr>
<th>point</th>
</tr>
</thead>
<tbody>
<tr>
<td>point({x: 2.3, y: 4.5, z: 2.0, crs: 'cartesian-3d'})</td>
</tr>
</tbody>
</table>

Rows: 1

### 10.12. LOAD CSV functions

LOAD CSV functions can be used to get information about the file that is processed by LOAD CSV.

The functions described on this page are only useful when run on a query that uses LOAD CSV. In all other contexts they will always return `null`.

**Functions:**

- `linenumber()`
- `file()`

#### 10.12.1. `linenumber()`

`linenumber()` returns the line number that LOAD CSV is currently using.

**Syntax:** `linenumber()`

**Returns:**
An Integer.

Considerations:

null will be returned if this function is called without a LOAD CSV context.

If the CSV file contains headers, the headers will be linenumber 1 and the 1st row of data will have a linenumber of 2.

10.12.2. file()

file() returns the absolute path of the file that LOAD CSV is using.

Syntax: file()

Returns:

A String.

Considerations:

null will be returned if this function is called without a LOAD CSV context.

10.13. User-defined functions

User-defined functions are written in Java, deployed into the database and are called in the same way as any other Cypher function.

There are two main types of functions that can be developed and used:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Usage</th>
<th>Developing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar</td>
<td>For each row the function takes parameters and returns a result</td>
<td>Using UDF</td>
<td>Extending Neo4j (UDF)</td>
</tr>
<tr>
<td>Aggregating</td>
<td>Consumes many rows and produces an aggregated result</td>
<td>Using aggregating UDF</td>
<td>Extending Neo4j (Aggregating UDF)</td>
</tr>
</tbody>
</table>

10.13.1. User-defined scalar functions

For each incoming row the function takes parameters and returns a single result.

This example shows how you invoke a user-defined function called join from Cypher.

Call a user-defined function

This calls the user-defined function org.neo4j.procedure.example.join().
MATCH (n:Member) RETURN org.neo4j.function.example.join(collect(n.name)) AS members

<table>
<thead>
<tr>
<th>members</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>John,Paul,George,Ringo</em></td>
</tr>
</tbody>
</table>

Rows: 1

For developing and deploying user-defined functions in Neo4j, see Extending Neo4j → User-defined functions.
Chapter 11. User-defined aggregation functions

Aggregating functions consume many rows and produce a single aggregated result.

This example shows how you invoke a user-defined aggregation function called `longestString` from Cypher.

11.1. Call a user-defined aggregation function

This calls the user-defined function `org.neo4j.function.example.longestString()`.

Query

```
MATCH (n:Member)
RETURN org.neo4j.function.example.longestString(n.name) AS member
```

Table 405. Result

<table>
<thead>
<tr>
<th>member</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;George&quot;</td>
</tr>
</tbody>
</table>

Rows: 1
Chapter 12. Administration

This section explains how to use Cypher to administer Neo4j databases, such as creating databases, managing indexes and constraints, and managing security.

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. Most administrative commands must be executed against the system database because they involve editing the metadata for the entire system. This includes all commands related to managing multiple databases as well as all commands for defining the security model: users, roles and privileges. The administrative commands that are specific to the schema of an individual database are still executed against that specific database. These include index and constraint management commands.

- **Databases**
  - Introduction
  - Listing databases
  - Creating databases
  - Stopping databases
  - Starting databases
  - Deleting databases

- **Indexes for search performance**
  - Introduction
  - Syntax
  - Composite index limitations
  - Examples

- **Indexes for full-text search**
  - Introduction
  - Procedures to manage full-text indexes
  - Create and configure full-text indexes
  - Query full-text indexes
  - Drop full-text indexes

- **Constraints**
  - Introduction
  - Syntax
  - Examples

- **Security**
  - Introduction
12.1. Databases

This section explains how to use Cypher to manage Neo4j databases: creating, deleting, starting and stopping individual databases within a single server.

- Introduction
- Listing databases
- Creating databases
- Stopping databases
- Starting databases
- Deleting databases

12.1.1. Introduction

Neo4j allows the same server to manage multiple databases. 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 need to be executing against the *system* database.

12.1.2. Listing databases

There are three different commands for listing databases. Listing all databases, listing a particular database or listing the default database.

All available databases can be seen using the command `SHOW DATABASES`.

Query

```
SHOW DATABASES
```

Table 406. Result

<table>
<thead>
<tr>
<th>name</th>
<th>address</th>
<th>role</th>
<th>requestedStatus</th>
<th>currentStatus</th>
<th>error</th>
<th>default</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;neo4j&quot;</td>
<td>&quot;localhost:7687&quot;</td>
<td>&quot;standalone&quot;</td>
<td>&quot;online&quot;</td>
<td>&quot;online&quot;</td>
<td>&quot;&quot;</td>
<td>true</td>
</tr>
<tr>
<td>&quot;system&quot;</td>
<td>&quot;localhost:7687&quot;</td>
<td>&quot;standalone&quot;</td>
<td>&quot;online&quot;</td>
<td>&quot;online&quot;</td>
<td>&quot;&quot;</td>
<td>false</td>
</tr>
</tbody>
</table>
Table 407. Result

<table>
<thead>
<tr>
<th>name</th>
<th>address</th>
<th>role</th>
<th>requestedStatus</th>
<th>currentStatus</th>
<th>error</th>
<th>default</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;system&quot;</td>
<td>&quot;localhost:7687&quot;</td>
<td>&quot;standalone&quot;</td>
<td>&quot;online&quot;</td>
<td>&quot;online&quot;</td>
<td>&quot;&quot;</td>
<td>false</td>
</tr>
</tbody>
</table>

Rows: 1

The default database can be seen using the command `SHOW DEFAULT DATABASE`.

Query

```sql
SHOW DEFAULT DATABASE
```

Table 408. Result

<table>
<thead>
<tr>
<th>name</th>
<th>address</th>
<th>role</th>
<th>requestedStatus</th>
<th>currentStatus</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;neo4j&quot;</td>
<td>&quot;localhost:7687&quot;</td>
<td>&quot;standalone&quot;</td>
<td>&quot;online&quot;</td>
<td>&quot;online&quot;</td>
<td>&quot;&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

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`.

12.1.3. Creating databases **Enterprise edition**

Databases can be created using `CREATE DATABASE`.

Note that the results of this command are filtered according to the `ACCESS` privileges the user has. However, a user with `CREATE/DROP DATABASE` or `DATABASE MANAGEMENT` privileges can see all databases regardless of their `ACCESS` privileges. If a user has not been granted `ACCESS` privilege to any databases, the command can still be executed but will only return the `system` database, which is always visible.

A particular database can be seen using the command `SHOW DATABASE name`.

Query

```sql
SHOW DATABASE system
```
CREATE DATABASE customers

0 rows, System updates: 1

Database names are subject to the standard Cypher restrictions on valid identifiers. The following naming rules apply:

- Database name length must be between 3 and 63 characters.
- The first character must be an ASCII alphabetic character.
- Subsequent characters can be ASCII alphabetic (mydatabase), numeric characters (mydatabase2), dots (main.db), and dashes (enclosed within backticks, e.g., CREATE DATABASE `main-db`).
- Names cannot end with dots or dashes.
- Names that begin with an underscore or with the prefix system are reserved for internal use.

When a database has been created, it will show up in the listing provided by the command SHOW DATABASES.

SHOW DATABASES

<table>
<thead>
<tr>
<th>name</th>
<th>address</th>
<th>role</th>
<th>requestedStatus</th>
<th>currentStatus</th>
<th>error</th>
<th>default</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;customers&quot;</td>
<td>&quot;localhost:7687&quot;</td>
<td>&quot;standalone&quot;</td>
<td>&quot;online&quot;</td>
<td>&quot;online&quot;</td>
<td>&quot;&quot;</td>
<td>false</td>
</tr>
<tr>
<td>&quot;neo4j&quot;</td>
<td>&quot;localhost:7687&quot;</td>
<td>&quot;standalone&quot;</td>
<td>&quot;online&quot;</td>
<td>&quot;online&quot;</td>
<td>&quot;&quot;</td>
<td>true</td>
</tr>
<tr>
<td>&quot;system&quot;</td>
<td>&quot;localhost:7687&quot;</td>
<td>&quot;standalone&quot;</td>
<td>&quot;online&quot;</td>
<td>&quot;online&quot;</td>
<td>&quot;&quot;</td>
<td>false</td>
</tr>
</tbody>
</table>

Rows: 3

This command is optionally idempotent, with the default behavior to throw an exception if the database already exists. Appending IF NOT EXISTS to the command will ensure that no exception is thrown 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.

CREATE DATABASE customers IF NOT EXISTS
CREATE OR REPLACE DATABASE customers

This is equivalent to running `DROP DATABASE customers IF EXISTS` followed by `CREATE DATABASE customers`.

The `IF NOT EXISTS` and `OR REPLACE` parts of this command cannot be used together.

### 12.1.4. Stopping databases

**Enterprise edition**

Databases can be stopped using the command `STOP DATABASE`.

**Query**

```sql
STOP DATABASE customers
```

0 rows, System updates: 1

The status of the stopped database can be seen using the command `SHOW DATABASE name`.

**Query**

```sql
SHOW DATABASE customers
```

<table>
<thead>
<tr>
<th>name</th>
<th>address</th>
<th>role</th>
<th>requestedStatus</th>
<th>currentStatus</th>
<th>error</th>
<th>default</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;customers&quot;</td>
<td>&quot;localhost:7687&quot;</td>
<td>&quot;standalone&quot;</td>
<td>&quot;offline&quot;</td>
<td>&quot;offline&quot;</td>
<td>&quot;&quot;</td>
<td>false</td>
</tr>
</tbody>
</table>

Table 410. Result

### 12.1.5. Starting databases

**Enterprise edition**

Databases can be started using the command `START DATABASE`.

**Query**

```sql
START DATABASE customers
```

0 rows, System updates: 1

The status of the started database can be seen using the command `SHOW DATABASE name`.

**Query**

```sql
SHOW DATABASE customers
```

Table 411. Result
12.1.6. Deleting databases **Enterprise edition**

Databases can be deleted using the command `DROP DATABASE`.

Query

```sql
DROP DATABASE customers
```

0 rows, System updates: 1

When a database has been deleted, it will no longer show up in the listing provided by the command `SHOW DATABASES`.

Query

```sql
SHOW DATABASES
```

Table 412. Result

<table>
<thead>
<tr>
<th>name</th>
<th>address</th>
<th>role</th>
<th>requestedStatus</th>
<th>currentStatus</th>
<th>error</th>
<th>default</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;neo4j&quot;</td>
<td>&quot;localhost:7687&quot;</td>
<td>&quot;standalone&quot;</td>
<td>&quot;online&quot;</td>
<td>&quot;online&quot;</td>
<td></td>
<td>true</td>
</tr>
<tr>
<td>&quot;system&quot;</td>
<td>&quot;localhost:7687&quot;</td>
<td>&quot;standalone&quot;</td>
<td>&quot;online&quot;</td>
<td>&quot;online&quot;</td>
<td></td>
<td>false</td>
</tr>
</tbody>
</table>

Rows: 2

This command is optionally idempotent, with the default behavior to throw an exception if the database does not exist. Appending `IF EXISTS` to the command will ensure that no exception is thrown and nothing happens should the database not exist.

Query

```sql
DROP DATABASE customers IF EXISTS
```

12.2. Indexes for search performance

This section explains how to manage indexes used for search performance.

- Introduction
- Syntax
This section describes how to manage indexes. For query performance purposes, it is important to also understand how the indexes are used by the Cypher planner. Refer to Query tuning for examples and in-depth discussions on how query plans result from different index and query scenarios. See specifically The use of indexes for examples of how various index scenarios result in different query plans.

For information on index configuration and limitations, refer to Operations Manual → Index configuration.

12.2.1. Introduction

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.

Cypher enables the creation of indexes on one or more properties for all nodes that have a given label:

- An index that is created on a single property for any given label is called a single-property index.
- An index that is created on more than one property for any given label is called a composite index.

Differences in the usage patterns between composite and single-property indexes are described in Composite index limitations.

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 will get an auto-generated name.
- The index name must be unique among both indexes and constraints.
- Index creation is not idempotent. An error will be thrown if you attempt to create the same index twice.

12.2.2. Syntax

Table 413. Syntax for managing indexes
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CREATE INDEX [index_name] FOR (n:LabelName) ON (n.propertyName)</code></td>
<td>Create a single-property index.</td>
<td>Best practice is to give the index a name when it is created. If the index is not explicitly named, it will get an auto-generated name. The index name must be unique among both indexes and constraints. Index creation is not idempotent. An error will be thrown if you attempt to create the same index twice.</td>
</tr>
<tr>
<td><code>CREATE INDEX [index_name] FOR (n:LabelName) ON (n.propertyName_1, n.propertyName_2, n.propertyName_n)</code></td>
<td>Create a composite index.</td>
<td></td>
</tr>
<tr>
<td><code>DROP INDEX index_name</code></td>
<td>Drop an index</td>
<td></td>
</tr>
<tr>
<td><code>CALL db.indexes</code></td>
<td>List all indexes in the database.</td>
<td></td>
</tr>
<tr>
<td><code>DROP INDEX ON :LabelName(propertyName)</code></td>
<td>Drop a single-property index without specifying a name.</td>
<td></td>
</tr>
<tr>
<td><code>DROP INDEX ON :LabelName (n.propertyName_1, n.propertyName_2, n.propertyName_n)</code></td>
<td>Drop a composite index without specifying a name.</td>
<td>This syntax is deprecated.</td>
</tr>
</tbody>
</table>

Creating an index requires the **CREATE INDEX** privilege, while dropping an index requires the **DROP INDEX** privilege. Listing indexes does not require any privileges.

Planner hints and the USING keyword describes how to make the Cypher planner use specific indexes (especially in cases where the planner would not necessarily have used them).

### 12.2.3. Composite index limitations

Like single-property indexes, composite indexes support all predicates:

- equality check: `n.prop = value`
- list membership check: `n.prop IN list`
- existence check: `exists(n.prop)`
- range search: `n.prop > value`
prefix search: STARTS WITH
suffix search: ENDS WITH
substring search: CONTAINS

For details about each operator, see Operators.

However, predicates might be planned as existence check and a filter. For most predicates, this can be avoided by following these restrictions:

- If there is any equality check and list membership check predicates, they need to be for the first properties defined by the index.
- There can be up to one range search or prefix search predicate.
- There can be any number of existence check predicates.
- Any predicate after a range search, prefix search or existence check predicate has to be an existence check predicate.

However, the suffix search and substring search predicates are always planned as existence check and a filter and any predicates following after will therefore also be planned as such.

For example, an index on :Label(prop1,prop2,prop3,prop4,prop5,prop6) and predicates:

```sql
WHERE n.prop1 = 'x' AND n.prop2 = 1 AND n.prop3 > 5 AND n.prop4 < 'e' AND n.prop5 = true AND exists(n.prop6)
```

will be planned as:

```sql
WHERE exists(n.prop1) AND exists(n.prop2) AND exists(n.prop3) AND exists(n.prop4) AND exists(n.prop5) AND exists(n.prop6)
```

with filters on `n.prop4 < 'e'` and `n.prop5 = true`, since `n.prop3` has a range search predicate.

And an index on :Label(prop1,prop2) with predicates:

```sql
WHERE n.prop1 ENDS WITH 'x' AND n.prop2 = false
```

will be planned as:

```sql
WHERE exists(n.prop1) AND exists(n.prop2)
```

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.
12.2.4. Examples

Create a single-property index

A named index on a single property for all nodes that have 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 will be created in the background.

Query

```
CREATE INDEX index_name FOR (n:Person) ON (n.surname)
```

Note that the index name needs to be unique.

Result

```
<table>
<thead>
<tr>
<th>No data returned.</th>
</tr>
</thead>
</table>
Indexes added: 1
```

12.3. Create a composite index

A named index on multiple properties for all nodes that have 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 labeled with the specified label and which contain all the properties in the index definition will be added to the index. Note that the composite index is not immediately available, but will be created in the background. The following statement will create a named composite index on all nodes labeled with `Person` and which have both an `age` and `country` property:

Query

```
CREATE INDEX index_name FOR (n:Person) ON (n.age, n.country)
```

Note that the index name needs to be unique.

Result

```
<table>
<thead>
<tr>
<th>No data returned.</th>
</tr>
</thead>
</table>
Indexes added: 1
```

12.4. Drop an index

An index on all nodes that have a label and property/properties combination can be dropped using the name with the `DROP INDEX index_name` command.

Query

```
DROP INDEX index_name
```
12.5. List indexes

Calling the built-in procedure `db.indexes` will list all indexes, including their names.

Query

```
CALL db.indexes
```

Result

```
+-------------------+
| No data returned. |
+-------------------+
Indexes removed: 1
```

12.6. Deprecated syntax

12.6.1. Drop a single-property index

An index on all nodes that have a label and single property combination can be dropped with `DROP INDEX ON :Label(property)`.

Query

```
DROP INDEX ON :Person(firstname)
```

Result

```
+-------------------+
| No data returned. |
+-------------------+
Indexes removed: 1
```

12.6.2. Drop a composite index

A composite index on all nodes that have a label and multiple property combination can be dropped with `DROP INDEX ON :Label(prop1, ..., propN)`. The following statement will drop a composite index on all
nodes labeled with Person and which have both an age and country property:

Query

```
DROP INDEX ON :Person(age, country)
```

Result

```
+-------------------+
<table>
<thead>
<tr>
<th>No data returned.</th>
</tr>
</thead>
</table>
Indexes removed: 1
```

12.7. Indexes for full-text search

This section describes how to use full-text indexes, to enable full-text search.

12.7.1. Introduction

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 btree indexes described in previous sections can only do exact matching or prefix matches on strings. A full-text index will instead tokenize the indexed string values, so it can match terms anywhere within the strings. How the indexed strings are tokenized and broken into terms, is determined by what analyzer the full-text index is configured with. For instance, the swedish analyzer knows how to tokenize and stem Swedish words, and will avoid indexing Swedish stop words. The complete list of stop words for each analyzer is included in the result of the `db.index.fulltext.listAvailableAnalyzers` procedure.

Full-text indexes:

- support the indexing of both nodes and relationships.
- support configuring custom analyzers, including analyzers that are not included with Lucene itself.
- can be queried using the Lucene query language.
- can return the score for each result from a query.
- are kept up to date automatically, as nodes and relationships are added, removed, and modified.
- will automatically populate newly created indexes with the existing data in a store.
- can be checked by the consistency checker, and they can be rebuilt if there is a problem with them.
- are a projection of the store, and can only index nodes and relationships by the contents of their properties.
- can support any number of documents in a single index.
- are created, dropped, and updated transactionally, and is automatically replicated throughout a cluster.
- can be accessed via Cypher procedures.
- can be configured to be eventually consistent, in which index updating is moved from the commit path.

346
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 btree indexes, a full-text index

- can be applied to more than one label.
- can be applied to relationship types (one or more).
- can be 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.

12.7.2. Procedures to manage full-text indexes

Full-text indexes are managed through built-in procedures, see Operations Manual → Procedures for a
complete reference.

The procedures for managing full-text indexes are listed in the table below:

<table>
<thead>
<tr>
<th>Usage</th>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create full-text node index</td>
<td><code>db.index.fulltext.createNodeIndex</code></td>
<td>Create a node fulltext index for the given labels and properties. The optional 'config' map parameter can be used to supply settings to the index. Supported settings are 'analyzer', for specifying what analyzer to use when indexing and querying. Use the <code>db.index.fulltext.listAvailableAnalyzers</code> procedure to see what options are available. And 'eventually_consistent' which can be set to 'true' to make this index eventually consistent, such that updates from committing transactions are applied in a background thread.</td>
</tr>
</tbody>
</table>
12.7.3. Create and configure full-text indexes

Full-text indexes are created with the `db.index.fulltext.createNodeIndex` and `db.index.fulltext.createRelationshipIndex` procedures. An index must be given a unique name when created, 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.

For instance, if we have a movie with a title.

**Query**

```
CREATE (m:Movie {title: "The Matrix"}) RETURN m.title
```

**Table 414. Result**

<table>
<thead>
<tr>
<th>m.title</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;The Matrix&quot;</td>
</tr>
</tbody>
</table>
And we have a full-text index on the `title` and `description` properties of movies and books.

Query

```sql
CALL db.index.fulltext.createNodeIndex("titlesAndDescriptions", ["Movie", "Book"], ["title", "description"])```

Then our movie node from above will be included in the index, even though it only has one of the indexed labels, and only one of the indexed properties:

Query

```sql
CALL db.index.fulltext.queryNodes("titlesAndDescriptions", "matrix")
```

<table>
<thead>
<tr>
<th>Table 415. Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>node.title</td>
</tr>
<tr>
<td>&quot;The Matrix&quot;</td>
</tr>
</tbody>
</table>

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 `db.index.fulltext.createNodeIndex` and `db.index.fulltext.createRelationshipIndex` procedures take an optional fourth argument, called `config`. The `config` parameter is a map from string to string, and can be used to set index-specific configuration settings. The `analyzer` setting can be used to configure an index-specific analyzer. The possible values for the `analyzer` setting can be listed with the `db.index.fulltext.listAvailableAnalyzers` procedure. The `eventually_consistent` setting, if set to "true", will put the index in an eventually consistent update mode. This means that updates will be applied in a background thread "as soon as possible", instead of during transaction commit like other indexes.

Query

```sql
CALL db.index.fulltext.createRelationshipIndex("taggedByRelationshipIndex", ["TAGGED_AS"], ["taggedByUser"], {analyzer: "url_or_email", 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 416. Result

| (empty result) |
| Rows: 0 |

12.7.4. Query full-text indexes

Full-text indexes will, in addition to any exact matches, also return approximate matches to a given query. Both the property values that are indexed, and the queries to the index, are processed through the analyzer such that the index can find that don’t exactly matches. The score that is returned alongside each result entry, represents how well the index thinks that entry matches the given query. The results are always returned in descending score order, where the best matching result entry is put first. To illustrate, in the example below, we search our movie database for "Full Metal Jacket", and even though there is an exact match as the first result, we also get three other less interesting results:

Query

```plaintext
CALL db.index.fulltext.queryNodes("titlesAndDescriptions", "Full Metal Jacket") YIELD node, score
RETURN node.title, score
```

Table 417. Result

<table>
<thead>
<tr>
<th>node.title</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Full Metal Jacket&quot;</td>
<td>1.4111186265945435</td>
</tr>
<tr>
<td>&quot;Full Moon High&quot;</td>
<td>0.44524085521698</td>
</tr>
<tr>
<td>&quot;The Jacket&quot;</td>
<td>0.3509606122970581</td>
</tr>
<tr>
<td>&quot;Yellow Jacket&quot;</td>
<td>0.3509606122970581</td>
</tr>
</tbody>
</table>

Rows: 4

Full-text indexes are powered by the Apache Lucene indexing and search library. This means that we can use Lucene’s full-text query language to express what we wish to search for. For instance, if we are only interested in exact matches, then we can quote the string we are searching for.

Query

```plaintext
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 418. Result

<table>
<thead>
<tr>
<th>node.title</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Full Metal Jacket&quot;</td>
<td>1.4111186265945435</td>
</tr>
</tbody>
</table>

Rows: 1

Lucene also allows us to use logical operators, such as AND and OR, to search for terms:
Only the **Full Metal Jacket** movie in our database has both the words *full* and *metal*.

Table 419. Result

<table>
<thead>
<tr>
<th>node.title</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Full Metal Jacket&quot;</td>
<td>1.1113792657852173</td>
</tr>
</tbody>
</table>

Rows: 1

It is also possible to search for only specific properties, by putting the property name and a colon in front of the text being searched for.

Query

```
CALL db.index.fulltext.queryNodes("titlesAndDescriptions", 'description:"surreal adventure"') YIELD node, score
RETURN node.title, node.description, score
```

Table 420. Result

<table>
<thead>
<tr>
<th>node.title</th>
<th>node.description</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Metallica Through The Never&quot;</td>
<td>&quot;The movie follows the young roadie Trip through his surreal adventure with the band.&quot;</td>
<td>0.2615291476249695</td>
</tr>
</tbody>
</table>

Rows: 1

A complete description of the Lucene query syntax can be found in the [Lucene documentation](#).  

12.7.5. Drop full-text indexes

A full-text node index is dropped by using the procedure `db.index.fulltext.drop`.

In the following example, we will drop the `taggedByRelationshipIndex` that we created previously:

Query

```
CALL db.index.fulltext.drop("taggedByRelationshipIndex")
```

Table 421. Result

<table>
<thead>
<tr>
<th>(empty result)</th>
</tr>
</thead>
</table>

Rows: 0

12.8. Constraints

*This section explains how to manage constraints used for ensuring data integrity.*
12.8.1. Introduction

The following constraint types are available:

**Unique node property constraints**

Unique property constraints ensure that property values are unique for all nodes with a specific label. Unique constraints do not mean that all nodes have to have a unique value for the properties — nodes without the property are not subject to this rule.

**Node property existence constraints**

Node property existence constraints ensure that a property exists for all nodes with a specific label. Queries that try to create new nodes of the specified label, but without this property, will fail. The same is true for queries that try to remove the mandatory property.

**Relationship property existence constraints**

Property existence constraints ensure that a property exists for all relationships with a specific type. All queries that try to create relationships of the specified type, but without this property, will fail. The same is true for queries that try to remove the mandatory property.

**Node key constraints**

Node key constraints ensure that, for a given label and set of properties:

i. All the properties exist on all the nodes with that label.

ii. The combination of the property values is unique.

Queries attempting to do any of the following will fail:

- Create new nodes without all the properties or where the combination of property values is not unique.
- Remove one of the mandatory properties.
- Update the properties so that the combination of property values is no longer unique.
Unique node property 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.

Creating a constraint has the following implications on indexes:

- Adding a unique property constraint on a property will also add a single-property index on that property, so such an index cannot be added separately.
- Adding a node key constraint for a set of properties will also add a composite index on those properties, so such an index cannot be added separately.
- Cypher will use these indexes for lookups just like other indexes. Refer to Indexes for search performance for more details on indexes.
- If a unique property constraint is dropped and the single-property index on the property is still required, the index will need to be created explicitly.
- If a node key constraint is dropped and the composite-property index on the properties is still required, the index will need to be created explicitly.

Additionally, the following is true for constraints:

- A given label can have multiple constraints, and unique and property existence constraints can be combined on the same property.
- Adding constraints is an atomic operation that can take a while — all existing data has to be scanned before Neo4j can turn the constraint 'on'.
- Best practice is to give the constraint a name when it is created. If the constraint is not explicitly named, it will get an auto-generated name.
- The constraint name must be unique among both indexes and constraints.
- Constraint creation is not idempotent. An error will be thrown if you attempt to create the same constraint twice.

12.8.2. Syntax

Table 422. Syntax for managing indexes
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE CONSTRAINT [constraint_name] ON (n:LabelName) ASSERT n.propertyName IS UNIQUE</td>
<td>Create a unique node property constraint.</td>
<td>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.</td>
</tr>
<tr>
<td>CREATE CONSTRAINT [constraint_name] ON (n:LabelName) ASSERT EXISTS (n.propertyName)</td>
<td>Create a node property existence constraint.</td>
<td></td>
</tr>
<tr>
<td>CREATE CONSTRAINT [constraint_name] ON ()-[<em>&quot;R:RELATIONSHIP_TYPE&quot;]</em>()-() ASSERT EXISTS (R.propertyName)</td>
<td>Create a relationship property existence constraint.</td>
<td>The constraint name must be unique among both indexes and constraints.</td>
</tr>
<tr>
<td>CREATE CONSTRAINT [constraint_name] ON (n:LabelName) ASSERT (n.propertyName_1, n.propertyName_2, … n.propertyName_n) IS NODE KEY</td>
<td>Create a node key constraint.</td>
<td>Constraint creation is not idempotent. An error will be thrown if you attempt to create the same constraint twice.</td>
</tr>
<tr>
<td>DROP CONSTRAINT constraint_name</td>
<td>Drop a constraint.</td>
<td></td>
</tr>
<tr>
<td>CALL db.constraints</td>
<td>List all constraints in the database.</td>
<td></td>
</tr>
<tr>
<td>DROP CONSTRAINT ON (n:LabelName) ASSERT n.propertyName IS UNIQUE</td>
<td>Drop a unique constraint without specifying a name.</td>
<td></td>
</tr>
<tr>
<td>DROP CONSTRAINT ON (n:LabelName) ASSERT EXISTS (n.propertyName)</td>
<td>Drop an exists constraint without specifying a name.</td>
<td></td>
</tr>
<tr>
<td>DROP CONSTRAINT ON ()-[<em>&quot;R:RELATIONSHIP_TYPE&quot;]</em>()-() ASSERT EXISTS (R.propertyName)</td>
<td>Drop a relationship property existence constraint without specifying a name.</td>
<td>This syntax is deprecated.</td>
</tr>
<tr>
<td>DROP CONSTRAINT ON (n:LabelName) ASSERT (n.propertyName_1, n.propertyName_2, … n.propertyName_n) IS NODE KEY</td>
<td>Drop a node key constraint without specifying a name.</td>
<td></td>
</tr>
</tbody>
</table>
Creating a constraint requires the CREATE CONSTRAINT privilege, while dropping a constraint requires the DROP CONSTRAINT privilege. Listing constraints does not require any privileges.

### 12.8.3. Examples

**Unique node property constraints**

**Create a unique constraint**

When creating a unique constraint, a name can be provided. The constraint ensures that your database will never contain more than one node with a specific label and one property value.

**Query**

```
CREATE CONSTRAINT constraint_name ON (book:Book) ASSERT book.isbn IS UNIQUE
```

**Result**

```
+-------------------+
| No data returned. |
+-------------------+
Unique constraints added: 1
```

**12.8.4. Create a node that complies with unique property constraints**

Create a Book node with an isbn that isn’t already in the database.

**Query**

```
CREATE (book:Book {isbn: '1449356265', title: 'Graph Databases'})
```

**Result**

```
+-------------------+
| No data returned. |
+-------------------+
Nodes created: 1
Properties set: 2
Labels added: 1
```

**12.8.5. Create a node that violates a unique property constraint**

Create a Book node with an isbn that is already used in the database.

**Query**

```
CREATE (book:Book {isbn: '1449356265', title: 'Graph Databases'})
```

In this case the node isn’t created in the graph.
Error message

Node(0) already exists with label `Book` and property `isbn` = '1449356265'

12.8.6. Failure to create a unique property constraint due to conflicting nodes

Create a unique property constraint on the property isbn on nodes with the Book label when there are two nodes with the same isbn.

Query

```
CREATE CONSTRAINT ON (book:Book) ASSERT book.isbn IS UNIQUE
```

In this case the constraint can’t be created because it is violated by existing data. We may choose to use Indexes for search performance instead or remove the offending nodes and then re-apply the constraint.

Error message

Unable to create CONSTRAINT ON ( book:Book ) ASSERT (book.isbn) IS UNIQUE: Both Node(0) and Node(1) have the label `Book` and property `isbn` = '1449356265'

12.9. Node property existence constraints

12.9.1. Create a node property existence constraint

When creating a node property existence constraint, a name can be provided. The constraint ensures that all nodes with a certain label have a certain property.

Query

```
CREATE CONSTRAINT constraint_name ON (book:Book) ASSERT exists(book.isbn)
```

Result

```
+-------------------+
| No data returned.  |
+-------------------+
Property existence constraints added: 1
```

12.9.2. Create a node that complies with property existence constraints

Create a Book node with an isbn property.

Query

```
CREATE (book:Book {isbn: '1449356265', title: 'Graph Databases'})
```
12.9.3. Create a node that violates a property existence constraint

Trying to create a Books node without an isbn property, given a property existence constraint on :Book(isbn).

Query

```plaintext
CREATE (book:Book {title: 'Graph Databases'})
```

In this case the node isn’t created in the graph.

Error message

Node(0) with label `Book` must have the property `isbn`

12.9.4. Removing an existence constrained node property

Trying to remove the isbn property from an existing node book, given a property existence constraint on :Book(isbn).

Query

```plaintext
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`

12.9.5. Failure to create a node property existence constraint due to existing node

Create a constraint on the property isbn on nodes with the Book label when there already exists a node without an isbn.

Query

```plaintext
CREATE CONSTRAINT ON (book:Book) ASSERT exists(book.isbn)
```

In this case the constraint can’t be created because it is violated by existing data. We may choose to remove the offending nodes and then re-apply the constraint.
Error message

Unable to create CONSTRAINT ON ( book:Book ) ASSERT exists(book.isbn):
Node(0) with label `Book` must have the property `isbn`.

12.10. Relationship property existence constraints

12.10.1. Create a relationship property existence constraint

When creating a relationship property existence constraint, a name can be provided. The constraint ensures all relationships with a certain type have a certain property.

Query

```
CREATE CONSTRAINT constraint_name ON ()-[like:LIKED]-() ASSERT exists(like.day)
```

Result

```
<table>
<thead>
<tr>
<th>No data returned.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property existence constraints added: 1</td>
</tr>
</tbody>
</table>
```

12.10.2. Create a relationship that complies with property existence constraints

Create a LIKED relationship with a day property.

Query

```
CREATE (user:User)-[like:LIKED {day: 'yesterday'}]->(book:Book)
```

Result

```
<table>
<thead>
<tr>
<th>No data returned.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes created: 2</td>
</tr>
<tr>
<td>Relationships created: 1</td>
</tr>
<tr>
<td>Properties set: 1</td>
</tr>
<tr>
<td>Labels added: 2</td>
</tr>
</tbody>
</table>
```

12.10.3. Create a relationship that violates a property existence constraint

Trying to create a LIKED relationship without a day property, given a property existence constraint :LIKED(day).
Query

```
CREATE (user:User)-[like:LIKED]->(book:Book)
```

In this case the relationship isn't created in the graph.

Error message

```
Relationship(0) with type `LIKED` must have the property `day`
```

12.10.4. Removing an existence constrained relationship property

Trying to remove the `day` property from an existing relationship `like` of type `LIKED`, given a property existence constraint `:LIKED(day)`.

Query

```
MATCH (user:User)-[like:LIKED]->(book:Book) REMOVE like.day
```

In this case the property is not removed.

Error message

```
Relationship(0) with type `LIKED` must have the property `day`
```

12.10.5. Failure to create a relationship property existence constraint due to existing relationship

Create a constraint on the property `day` on relationships with the `LIKED` type when there already exists a relationship without a property named `day`.

Query

```
CREATE CONSTRAINT ON ()-[like:LIKED]-() ASSERT exists(like.day)
```

In this case the constraint can’t be created because it is violated by existing data. We may choose to remove the offending relationships and then re-apply the constraint.

Error message

```
Unable to create CONSTRAINT ON ()-[ liked:LIKED ]-() ASSERT exists(liked.day):
Relationship(0) with type `LIKED` must have the property `day`
```

12.11. Node key constraints [Enterprise edition]

12.11.1. Create a node key constraint

When creating a node key constraint, a name can be provided. The constraint ensures that all nodes with a particular label have a set of defined properties whose combined value is unique and all properties in the
set are present.

Query

```
CREATE CONSTRAINT constraint_name ON (n:Person) ASSERT (n.firstname, n.surname) IS NODE KEY
```

Result

```
+-------------------+
| No data returned. |
+-------------------+
Node key constraints added: 1
```

12.11.2. Create a node that complies with node key constraints

Create a `Person` node with both a `firstname` and `surname` property.

Query

```
CREATE (p:Person {firstname: 'John', surname: 'Wood', age: 55})
```

Result

```
+-------------------+
| No data returned. |
+-------------------+
Nodes created: 1
Properties set: 3
Labels added: 1
```

12.11.3. Create a node that violates a node key constraint

Trying to create a `Person` node without a `surname` property, given a node key constraint on :Person(firstname, surname), will fail.

Query

```
CREATE (p:Person {firstname: 'Jane', age: 34})
```

In this case the node isn't created in the graph.

Error message

```
Node(0) with label `Person` must have the properties (firstname, surname)
```

12.11.4. Removing a NODE KEY-constrained property

Trying to remove the `surname` property from an existing node `Person`, given a NODE KEY constraint on :Person(firstname, surname).
Query

MATCH (p:Person { firstname: 'John', surname: 'Wood'}) REMOVE p.surname

In this case the property is not removed.

Error message

Node(0) with label `Person` must have the properties (firstname, surname)

12.11.5. Failure to create a node key constraint due to existing node

Trying to create a node key constraint on the property surname on nodes with the Person label will fail when a node without a surname already exists in the database.

Query

CREATE CONSTRAINT ON (n:Person) ASSERT (n.firstname, n.surname) IS NODE KEY

In this case the node key constraint can’t be created because it is violated by existing data. We may choose to remove the offending nodes and then re-apply the constraint.

Error message

Unable to create CONSTRAINT ON ( person:Person ) ASSERT exists(person.firstname, person.surname): Node(0) with label `Person` must have the properties (firstname, surname)

12.11.6. Drop a constraint

A constraint can be dropped using the name with the DROP CONSTRAINT constraint_name command. It is the same command for unique property, property existence and node key constraints.

Query

DROP CONSTRAINT constraint_name

Result

+-------------------+
<table>
<thead>
<tr>
<th>No data returned.</th>
</tr>
</thead>
</table>
Named constraints removed: 1

12.11.7. List constraints

Calling the built-in procedure db.constraints will list all constraints, including their names.

Query

CALL db.constraints
### 12.12. Deprecated syntax

#### 12.12.1. Drop a unique constraint

By using `DROP CONSTRAINT`, you remove a constraint from the database.

**Query**

```
DROP CONSTRAINT ON (book:Book) ASSERT book.isbn IS UNIQUE
```

**Result**

```
<table>
<thead>
<tr>
<th>name</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;constraint_ca412c3d&quot;</td>
<td>&quot;CONSTRAINT ON ( book:Book ) ASSERT (book.isbn) IS UNIQUE&quot;</td>
</tr>
</tbody>
</table>
```

No data returned.

Unique constraints removed: 1

#### 12.12.2. Drop a node property existence constraint

By using `DROP CONSTRAINT`, you remove a constraint from the database.

**Query**

```
DROP CONSTRAINT ON (book:Book) ASSERT exists(book.isbn)
```

**Result**

```
<table>
<thead>
<tr>
<th>name</th>
<th>description</th>
</tr>
</thead>
</table>
```

No data returned.

Property existence constraints removed: 1

#### 12.12.3. Drop a relationship property existence constraint

To remove a constraint from the database, use `DROP CONSTRAINT`.

**Query**

```
DROP CONSTRAINT ON ()-[like:LIKED]-() ASSERT exists(like.day)
```
12.12.4. Drop a node key constraint

Use `DROP CONSTRAINT` to remove a node key constraint from the database.

**Query**

```
DROP CONSTRAINT ON (n:Person) ASSERT (n.firstname, n.surname) IS NODE KEY
```

**Result**

```
| No data returned. |
+-------------------+
Property existence constraints removed: 1
```

12.13. Security

*This section explains how to use Cypher to manage Neo4j role-based access control and fine-grained security.*

- Introduction
- Syntax summaries
- User and role management
  - User management
    - Listing users
    - Creating users
    - Modifying users
    - Changing the current user’s password
    - Deleting users
  - Role management
    - Listing roles
    - Creating roles
    - Deleting roles
    - Assigning roles
    - Revoking roles
- Graph and sub-graph access control
The GRANT, DENY and REVOKE commands

- Listing privileges
- The TRAVERSE privilege
- The READ privilege
- The MATCH privilege
- The WRITE privilege
- The REVOKE command

- Security of administration
  - The admin role
  - Database administration
    - The database ACCESS privilege
    - The database START/STOP privileges
    - The INDEX MANAGEMENT privileges
    - The CONSTRAINT MANAGEMENT privileges
    - The NAME MANAGEMENT privileges
    - Granting all database administration privileges
  - DBMS administration
    - Creating custom roles with DBMS privileges
    - The dbms ROLE MANAGEMENT privileges

- Known limitations of security
  - Security and indexes
  - Security and labels
  - Security and count store operations

12.13.1. Introduction

This section introduces the sections on how to manage Neo4j role-based access control and fine-grained security.

Neo4j has a complex security model stored in the system graph, maintained in a special database called the system database. All administrative commands need to be executing against the system database. For more information on how to manage multiple databases, refer to the section on administering databases. Neo4j 3.1 introduced the concept of role-based access control. It was possible to create users and assign them to roles to control whether the users could 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. Instead they are defined in terms of their underlying privileges and they can be modified by adding an removing these access rights. In addition any
new roles can be assigned any combination of privileges to create the specific access control desired. A major additional capability is sub-graph access control whereby read-access to the graph can be limited to specific combinations of label, relationship-type and property.

12.13.2. Syntax summaries

Almost all administration commands have variations in the commands. Parts of the command that are optional or can have multiple values are most common. To show all versions of a command, a summary of the syntax will be presented. These summaries will use some special characters to indicate such variations.

The special characters and their meaning are as follows:

Table 423. Special characters in syntax summaries

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Or; used to indicate alternative parts of a command. Needs to be part of a grouping.</td>
<td>If the syntax needs to specify either a name or *, this can be indicated with *</td>
</tr>
<tr>
<td>{ and }</td>
<td>Used to group parts of the command, common together with</td>
<td>.</td>
</tr>
<tr>
<td>[ and ]</td>
<td>Used to indicate an optional part of the command. It also groups alternatives together, when there can be either of the alternatives or nothing.</td>
<td>If a keyword in the syntax can either be in singular or plural, we can indicate that the is optional with GRAPH[S].</td>
</tr>
<tr>
<td>...</td>
<td>Repeated pattern, the command part immediately before this is repeated.</td>
<td>A comma separated list of names would be name[, ...].</td>
</tr>
<tr>
<td>&quot;</td>
<td>When a special character is part of the syntax itself, we surround it with &quot; to indicate this.</td>
<td>To include { in the syntax use &quot;(* { *</td>
</tr>
</tbody>
</table>

The special characters in the table above are the only ones that need to be escaped using " in the syntax summaries.

An example that uses all special characters is granting the READ privilege:

```
GRANT READ
"(* { * | property[, ...] } )"
ON GRAPH[S] ( * | name )
[ 
    ELEMENT[S] ( * | label-or-rel-type[, ...] )
| NODE[S] ( * | label[, ...] )
| RELATIONSHIP[S] ( * | rel-type[, ...] )
]
TO role[, ...]
```

Some things to notice about this command is that it includes { and } in the syntax, and between them has
a grouping of either a list 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.

In difference, there is no need to escape any characters in the node property existence constraint creation command. This is because ( and ) are not special characters, and the [ and ] indicate that the constraint name is optional, and are not part of the command.

```cypher
cREATE CONSTRAINT [constraint_name]
on (n:LabelName)
ASSERT EXISTS (n.propertyName)
```

12.13.3. Security

This section explains how to use Cypher to manage Neo4j role-based access control and fine-grained security.

- Introduction
- Syntax summaries
- User and role management
  - User management
    - Listing users
    - Creating users
    - Modifying users
    - Changing the current user’s password
    - Deleting users
  - Role management
    - Listing roles
    - Creating roles
    - Deleting roles
    - Assigning roles
    - Revoking roles
- Graph and sub-graph access control
  - The GRANT, DENY and REVOKE commands
  - Listing privileges
  - The TRAVERSE privilege
  - The READ privilege
  - The MATCH privilege
  - The WRITE privilege
The REVOKE command

- Security of administration
  - The admin role
  - Database administration
    - The database ACCESS privilege
    - The database START/STOP privileges
    - The INDEX MANAGEMENT privileges
    - The CONSTRAINT MANAGEMENT privileges
    - The NAME MANAGEMENT privileges
    - Granting all database administration privileges
  - DBMS administration
    - Creating custom roles with DBMS privileges
    - The dbms ROLE MANAGEMENT privileges
- Known limitations of security
  - Security and indexes
  - Security and labels
  - Security and count store operations

Introduction

This section introduces the sections on how to manage Neo4j role-based access control and fine-grained security.

Neo4j has a complex security model stored in the system graph, maintained in a special database called the system database. All administrative commands need to be executing against the system database. For more information on how to manage multiple databases, refer to the section on administering databases. Neo4j 3.1 introduced the concept of role-based access control. It was possible to create users and assign them to roles to control whether the users could 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. Instead they are defined in terms of their underlying privileges and they can be modified by adding an removing these access rights. In addition any new roles created can by assigned any combination of privileges to create the specific access control desired. A major additional capability is sub-graph access control whereby read-access to the graph can be limited to specific combinations of label, relationship-type and property.

Syntax summaries

Almost all administration commands have variations in the commands. Parts of the command that are optional or can have multiple values are most common. To show all versions of a command, a summary of
the syntax will be presented. These summaries will use some special characters to indicate such variations.

The special characters and their meaning are as follows:

Table 424. Special characters in syntax summaries

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Or; used to indicate alternative parts of a command. Needs to be part of a grouping.</td>
<td>If the syntax needs to specify either a name or *, this can be indicated with *</td>
</tr>
<tr>
<td>{ and }</td>
<td>Used to group parts of the command, common together with</td>
<td>To use the or in the syntax summary it needs to be in a group, [*</td>
</tr>
<tr>
<td>[ and ]</td>
<td>Used to indicate an optional part of the command. It also groups alternatives together, when there can be either of the alternatives or nothing.</td>
<td>If a keyword in the syntax can either be in singular or plural, we can indicate that the S is optional with GRAPH[S].</td>
</tr>
<tr>
<td>…</td>
<td>Repeated pattern, the command part immediately before this is repeated.</td>
<td>A comma separated list of names would be name[, …].</td>
</tr>
<tr>
<td>&quot;</td>
<td>When a special character is part of the syntax itself, we surround it with &quot; to indicate this.</td>
<td>To include { in the syntax use &quot;(* { *</td>
</tr>
</tbody>
</table>

The special characters in the table above are the only ones that need to be escaped using " in the syntax summaries.

An example that uses all special characters is granting the READ privilege:

```
grant read
  "(* { * | property[, ...] } ")"
  on graph[S] { * | name }
  [  
    element[S] ( * | label-or-rel-type[, ...] )
    | node[S] { * | label[, ...] }
    | relationship[S] { * | rel-type[, ...] }
  ]
  to role[, ...]
```

Some things to notice about this command is that it includes { and } in the syntax, and between them has a grouping of either a list 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.

In difference, there is no need to escape any characters in the node property existence constraint creation command. This is because ( and ) are not special characters, and the [ and ] indicate that the constraint name is optional, and are not part of the command.
12.13.4. User and role management

This section explains how to use Cypher to manage Neo4j role-based access control through users and roles.

- User Management
  - Listing users
  - Creating users
  - Modifying users
  - Changing the current user’s password
  - Deleting users
- Role management
  - Listing roles
  - Creating roles
  - Deleting roles
  - Assigning roles
  - Revoking roles

User Management

Users can be created and managed using a set of Cypher administration commands executed against the system database.

Table 425. User management command syntax

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Type of user</th>
<th>Available in Community Edition</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHOW USERS</td>
<td>List all users</td>
<td>Admin</td>
<td>+</td>
</tr>
<tr>
<td>SHOW USER name PRIVILEGES</td>
<td>List the privileges granted to a user</td>
<td>Admin</td>
<td>-</td>
</tr>
</tbody>
</table>
### Command

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Type of user</th>
<th>Available in Commuity Edition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE [OR REPLACE] USER name [IF NOT EXISTS] SET PASSWORD 'password' [SET PASSWORD] CHANGE [NOT] REQUIRED [SET STATUS (ACTIVE</td>
<td>SUSPENDED)]</td>
<td>Create a new user</td>
<td>Admin</td>
</tr>
<tr>
<td>ALTER USER name SET { PASSWORD 'password' [SET PASSWORD] CHANGE [NOT] REQUIRED [SET STATUS (ACTIVE</td>
<td>SUSPENDED)]</td>
<td>Modify the settings for an existing user</td>
<td>Admin</td>
</tr>
<tr>
<td>ALTER CURRENT USER SET PASSWORD FROM 'oldPassword' TO 'newPassword'</td>
<td>Change the current user’s password</td>
<td>Normal user</td>
<td>-</td>
</tr>
<tr>
<td>DROP USER name [IF EXISTS]</td>
<td>Drop (remove) an existing user</td>
<td>Admin</td>
<td>+</td>
</tr>
</tbody>
</table>

### Listing users

Available users can be seen using `SHOW USERS` which will produce a table of users with four columns:

Table 426. List users output

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Available in Community Edition</th>
</tr>
</thead>
<tbody>
<tr>
<td>user</td>
<td>user name</td>
<td>+</td>
</tr>
<tr>
<td>roles</td>
<td>roles granted to the user</td>
<td>-</td>
</tr>
<tr>
<td>passwordChangeRequired</td>
<td>if true the user must change their password at next login</td>
<td>+</td>
</tr>
<tr>
<td>suspended</td>
<td>if true the user is currently suspended (cannot log in)</td>
<td>-</td>
</tr>
</tbody>
</table>

Query

```
SHOW USERS
```

Table 427. Result

370
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 first login.

<table>
<thead>
<tr>
<th>user</th>
<th>roles</th>
<th>passwordChangeRequired</th>
<th>suspended</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;neo4j&quot;</td>
<td>[&quot;admin&quot;]</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

Rows: 1

The SHOW USER name PRIVILEGES command is described in Listing privileges.

Creating users

Users can be created using CREATE USER.

Command syntax

```
CREATE [OR REPLACE] USER name [IF NOT EXISTS]
SET PASSWORD 'password'
[[SET PASSWORD] CHANGE [NOT] REQUIRED]
[SET STATUS {ACTIVE | SUSPENDED}]
```

If the optional SET PASSWORD CHANGE [NOT] REQUIRED is omitted then the default is CHANGE REQUIRED. The default for SET STATUS is ACTIVE. The password can either be a string value or a string parameter.

For example, we can create the user jake in a suspended state and the requirement to change his password.

Query

```
CREATE USER jake SET PASSWORD 'abc' CHANGE REQUIRED SET STATUS SUSPENDED
```

0 rows, System updates: 1

The SUSPENDED flag is an enterprise feature.

The created user will appear on the list provided by SHOW USERS.

Query

```
SHOW USERS
```

Table 428. Result

<table>
<thead>
<tr>
<th>user</th>
<th>roles</th>
<th>passwordChangeRequired</th>
<th>suspended</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;neo4j&quot;</td>
<td>[&quot;admin&quot;]</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>&quot;jake&quot;</td>
<td>[]</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

Rows: 2
When creating a user, their initial list of roles is empty. In Neo4j community-edition this is not important as all users have implied administrator privileges. In Neo4j enterprise-edition a new user like this will have no rights and it is important to grant roles to the user.

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 command will ensure that no exception is thrown and nothing happens should the user already exist. Adding `OR REPLACE` to the command will result in any existing user being deleted and a new one created.

```
CREATE USER jake IF NOT EXISTS SET PASSWORD 'xyz'
```

0 rows

```
CREATE OR REPLACE USER jake SET PASSWORD 'xyz'
```

0 rows, System updates: 2

This is equivalent to running `DROP USER jake IF EXISTS` followed by `CREATE USER jake SET PASSWORD 'xyz'.`

```
CREATE OR REPLACE USER jake SET PASSWORD 'xyz'
```

The `IF NOT EXISTS` and `OR REPLACE` parts of this command cannot be used together.

Modifying users **Enterprise edition**

Users can be modified using `ALTER USER`.

**Command syntax**

```
ALTER USER name SET {
  PASSWORD 'password'
  [[SET PASSWORD] CHANGE [NOT] REQUIRED]
  [SET STATUS {ACTIVE | SUSPENDED}] |
  PASSWORD CHANGE [NOT] REQUIRED
  [SET STATUS {ACTIVE | SUSPENDED}] |
  STATUS {ACTIVE | SUSPENDED}
}
```

The `password` can either be a string value or a string parameter.

For example, we can modify the user `jake` with a new password and active status as well as remove the requirement to change his password.

```
ALTER USER jake SET PASSWORD 'abc123' CHANGE NOT REQUIRED SET STATUS ACTIVE
```
When altering a user it is only necessary to specify the changes required. For example, leaving out any `STATUS` change part of the query will leave that unchanged.

The changes to the user will appear on the list provided by `SHOW USERS`.

### Query

```
SHOW USERS
```

#### Table 429. Result

<table>
<thead>
<tr>
<th>user</th>
<th>roles</th>
<th>passwordChangeRequired</th>
<th>suspended</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;neo4j&quot;</td>
<td>[&quot;admin&quot;]</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>&quot;jake&quot;</td>
<td>[]</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

### Changing the current user’s password

Users can change their own 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.

### Query

```
ALTER CURRENT USER SET PASSWORD FROM 'abc123' TO '123xyz'
```

### Deleting users

Users can be deleted using `DROP USER`.

### Query

```
DROP USER jake
```

When a user has been deleted, it will no longer appear on the list provided by `SHOW USERS`.
Query

SHOW USERS

Table 430. Result

<table>
<thead>
<tr>
<th>user</th>
<th>roles</th>
<th>passwordChangeRequired</th>
<th>suspended</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;neo4j&quot;</td>
<td>[&quot;admin&quot;]</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

Rows: 1

This command is optionally idempotent, with the default behavior to throw an exception if the user does not exist. Appending IF EXISTS to the command will ensure that no exception is thrown and nothing happens should the user not exist.

Query

```
DROP USER jake IF EXISTS
```

0 rows

Role Management **Enterprise edition**

Roles can be created and managed using a set of Cypher administration commands executed against the system database.

The role name **PUBLIC** is not permitted. This is a system-reserved role, and cannot be used.

Table 431. Role management command syntax

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`SHOW [ALL</td>
<td>POPULATED] ROLES [WITH USERS]`</td>
</tr>
<tr>
<td><code>SHOW ROLE name PRIVILEGES</code></td>
<td>List the privileges granted to a role</td>
</tr>
<tr>
<td><code>CREATE [OR REPLACE] ROLE name [IF NOT EXISTS] [AS COPY OF name]</code></td>
<td>Create a new role</td>
</tr>
<tr>
<td><code>DROP ROLE name [IF EXISTS]</code></td>
<td>Drop (remove) an existing role</td>
</tr>
<tr>
<td><code>GRANT ROLE[S] name[, ...] TO user[, ...]</code></td>
<td>Assign one or multiple roles to one or multiple users</td>
</tr>
</tbody>
</table>
REVOKE ROLE[S] name[, ...] FROM user[, ...]

Remove one or multiple roles from one or multiple users

Listing roles

Available roles can be seen using SHOW ROLES.

Query

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:

- **reader** - can perform traverse and read operations on all databases except system.
- **editor** - can perform traverse, read, and write operations on all databases except system, but cannot make 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, users, roles, and privileges.

More information about the built-in roles can be found in [Operations Manual → Built-in roles](#).

<table>
<thead>
<tr>
<th>role</th>
<th>isBuiltIn</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;admin&quot;</td>
<td>true</td>
</tr>
<tr>
<td>&quot;publisher&quot;</td>
<td>true</td>
</tr>
<tr>
<td>&quot;editor&quot;</td>
<td>true</td>
</tr>
<tr>
<td>&quot;reader&quot;</td>
<td>true</td>
</tr>
<tr>
<td>&quot;architect&quot;</td>
<td>true</td>
</tr>
</tbody>
</table>

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 appended to the commands. This will give one result row for each user, so if a role is assigned to two users then it will show up twice in the result.

Query

SHOW POPULATED ROLES WITH USERS

The table of results will show information about the role and what database it belongs to.
The **SHOW ROLE** name PRIVILEGES command is found in Listing privileges.

Creating roles **Enterprise edition**

Roles can be created using **CREATE ROLE**.

Query

```sql
CREATE ROLE myrole
```

0 rows, System updates: 1

The following naming rules apply:
- The first character must be an ASCII alphabetic character.
- Subsequent characters can be ASCII alphabetic, numeric characters, and underscore.

A role can also be copied, keeping its privileges, using **CREATE ROLE AS COPY OF**.

Query

```sql
CREATE ROLE mysecondrole AS COPY OF myrole
```

0 rows, System updates: 1

The created roles will appear on the list provided by **SHOW ROLES**.

Query

```sql
SHOW ROLES
```

Table 434. Result

<table>
<thead>
<tr>
<th>role</th>
<th>isBuiltIn</th>
<th>member</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;admin&quot;</td>
<td>true</td>
<td>&quot;neo4j&quot;</td>
</tr>
<tr>
<td>&quot;publisher&quot;</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>&quot;editor&quot;</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>&quot;reader&quot;</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>&quot;architect&quot;</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>&quot;myrole&quot;</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>&quot;mysecondrole&quot;</td>
<td>false</td>
<td></td>
</tr>
</tbody>
</table>

Rows: 1
These command versions are optionally idempotent, with the default behavior to throw an exception if the role already exists. Appending `IF NOT EXISTS` to the command will ensure that no exception is thrown and nothing happens should the role already exist. Adding `OR REPLACE` to the command will result in any existing role being deleted and a new one created.

```
CREATE ROLE myrole IF NOT EXISTS
```

0 rows

```
CREATE OR REPLACE ROLE myrole
```

0 rows, System updates: 2

This is equivalent to running `DROP ROLE myrole IF EXISTS` followed by `CREATE ROLE myrole`.

The `IF NOT EXISTS` and `OR REPLACE` parts of this command cannot be used together.

Deleting roles

Roles can be deleted using `DROP ROLE` command.

```
DROP ROLE mysecondrole
```

0 rows, System updates: 1

When a role has been deleted, it will no longer appear on the list provided by `SHOW ROLES`.

<table>
<thead>
<tr>
<th>role</th>
<th>isBuiltIn</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;admin&quot;</td>
<td>true</td>
</tr>
<tr>
<td>&quot;publisher&quot;</td>
<td>true</td>
</tr>
<tr>
<td>&quot;editor&quot;</td>
<td>true</td>
</tr>
<tr>
<td>&quot;reader&quot;</td>
<td>true</td>
</tr>
</tbody>
</table>
This command is optionally idempotent, with the default behavior to throw an exception if the role does not exist. Appending \texttt{IF EXISTS} to the command will ensure that no exception is thrown and nothing happens should the role not exist.

Query

\begin{verbatim}
DROP ROLE mysecondrole IF EXISTS
\end{verbatim}

0 rows

Assigning roles to users \textbf{Enterprise edition}

Users can be given access rights by assigning them roles using \texttt{GRANT ROLE}.

Query

\begin{verbatim}
GRANT ROLE myrole TO jake
\end{verbatim}

0 rows, System updates: 1

The roles assigned to each user can be seen in the list provided by \texttt{SHOW USERS}.

Query

\begin{verbatim}
SHOW USERS
\end{verbatim}

Table 436. Result

\begin{tabular}{|c|c|c|c|}
\hline
user & roles & passwordChangeRequired & suspended \\
\hline
"neo4j" & ["admin"] & true & false \\
"jake" & ["myrole"] & false & false \\
"user1" & [] & true & false \\
"user2" & [] & true & false \\
"user3" & [] & true & false \\
\hline
\end{tabular}

Rows: 5

It is possible to assign multiple roles to multiple users in one command.

Query

\begin{verbatim}
GRANT ROLES role1, role2 TO user1, user2, user3
\end{verbatim}

0 rows, System updates: 6
Revoking roles from users

Users can lose access rights by revoking roles from them using \texttt{REVOKE ROLE}.

Query

\begin{verbatim}
REVOKE ROLE myrole FROM jake
\end{verbatim}

0 rows, System updates: 1

The roles revoked from users can no longer be seen in the list provided by \texttt{SHOW USERS}.

Query

\begin{verbatim}
SHOW USERS
\end{verbatim}

Table 438. Result

<table>
<thead>
<tr>
<th>user</th>
<th>roles</th>
<th>passwordChangeRequired</th>
<th>suspended</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;neo4j&quot;</td>
<td>[&quot;admin&quot;]</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>&quot;jake&quot;</td>
<td>[&quot;myrole&quot;]</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>&quot;user1&quot;</td>
<td>[&quot;role1&quot;,&quot;role2&quot;]</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>&quot;user2&quot;</td>
<td>[&quot;role2&quot;,&quot;role1&quot;]</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>&quot;user3&quot;</td>
<td>[&quot;role2&quot;,&quot;role1&quot;]</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

Rows: 5

It is possible to revoke multiple roles from multiple users in one command.

Query

\begin{verbatim}
REVOKE ROLES role1, role2 FROM user1, user2, user3
\end{verbatim}
12.13.5. Graph and sub-graph access control

This section explains how to use Cypher to manage privileges for Neo4j role-based access control and fine-grained security.

- The **GRANT**, **DENY** and **REVOKE** commands
- Listing privileges
- The **TRAVERSE** privilege
- The **READ** privilege
- The **MATCH** privilege
- The **WRITE** privilege
- The **REVOKE** command

Privileges control the access rights to graph elements using a combined whitelist/blacklist mechanism. It is possible to grant access, or deny access, or a combination of the two. The user will be able to access the resource if they have a grant (whitelist) and do not have a deny (blacklist) relevant to that resource. All other combinations of **GRANT** and **DENY** will result in the matching path being invisible. It will appear to the user as if they have a smaller database (smaller graph).

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 **GRANT**, **DENY** and **REVOKE** commands **Enterprise edition**

The **GRANT** command allows an administrator to grant a privilege to a role in order to access an entity. The **DENY** command allows an administrator to deny a privilege to a role in order to prevent access to an entity. The **REVOKE** command allows an administrator to remove a previously granted or denied privilege. The syntax is:

Table 439. General graph privilege command syntax

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GRANT</strong> graph-privilege ON GRAPH[S] {*</td>
<td>name} [entity] TO role[, ...]</td>
</tr>
<tr>
<td><strong>DENY</strong> graph-privilege ON GRAPH[S] {*</td>
<td>name} [entity] TO role[, ...]</td>
</tr>
</tbody>
</table>
**Command**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REVOKE GRANT graph-privilege ON GRAPH[S] [*</td>
<td>name] [entity] FROM role[, ...]</td>
</tr>
<tr>
<td>REVOKE DENY graph-privilege ON GRAPH[S] [*</td>
<td>name] [entity] FROM role[, ...]</td>
</tr>
<tr>
<td>REVOKE graph-privilege ON GRAPH[S] [*</td>
<td>name] [entity] FROM role[, ...]</td>
</tr>
</tbody>
</table>

Where the components are:

- **graph-privilege**
  - **TRAVERSE**
    - allows the specified entities to be found
  - **READ**
    - allows the specified properties to be read on the found entities. Note that granting property READ access does not imply that the entities with that property can be found. For example if there is also a DENY on it, the entity will not be found. The props can be * which means all properties.
  - **MATCH**
    - this combines both TRAVERSE and READ allowing an entity to be found and its properties read.
  - **WRITE**
    - this privilege can only be assigned to all nodes, relationships and properties in the entire graph (this means that the entity part of the command must also be ELEMENTS +**+ and cannot be more specific).

- **name**
  - The graph or graphs to associate the privilege with. In 4.0 there can be only one graph per database, and therefore this command uses the database name to refer to that graph. Note that if you delete a database and create a new one with the same name, the new one will NOT have any of the privileges specifically assigned to the deleted graph.
  - It can be * which means all graphs. Any new databases created after this command will also be associated with these privileges.

- **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 * which means all labels or types.
- Multiple labels or types can be specified, comma-separated.
- Defaults to **ELEMENTS** * if omitted.

- **role[, ...]**
  - The role or roles to associate the privilege with, comma-separated.

It is important to note that using DENY does NOT erase a GRANT command; they both exist. The only way to erase a privilege is with REVOKE.

**Figure 1.** GRANT and DENY Syntax. The { and } are part of the syntax and not used for grouping.

The below image shows the hierarchy between the different graph privileges.

**Figure 2.** Graph privileges hierarchy

**Listing privileges** **Enterprise edition**

Available privileges for all roles can be seen using **SHOW PRIVILEGES**.

**Query**

```
SHOW PRIVILEGES
```

Lists all privileges for all roles. The table contains columns describing the privilege:

- **access**: whether the privilege is granted or denied (whitelist or blacklist)
- **action**: which type of privilege this is: access, traverse, read, write, token, schema or admin
- **resource**: what type of scope this privilege applies to: the entire dbms, a database, a graph or sub-graph access
- **graph**: the specific database or graph this privilege applies to
- **segment**: for sub-graph access control, this describes the scope in terms of labels or relationship types
• role: the role the privilege is granted to

<table>
<thead>
<tr>
<th>access</th>
<th>action</th>
<th>resource</th>
<th>graph</th>
<th>segment</th>
<th>role</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;read&quot;</td>
<td>&quot;all_properties&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;NODE(*)&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;write&quot;</td>
<td>&quot;all_properties&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;NODE(*)&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;traverse&quot;</td>
<td>&quot;graph&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;NODE(*)&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;read&quot;</td>
<td>&quot;all_properties&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;RELATIONSHIP(*)&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;write&quot;</td>
<td>&quot;all_properties&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;RELATIONSHIP(*)&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;traverse&quot;</td>
<td>&quot;graph&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;RELATIONSHIP(*)&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;access&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;admin&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;schema&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;token&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;read&quot;</td>
<td>&quot;all_properties&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;NODE(*)&quot;</td>
<td>&quot;architect&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;write&quot;</td>
<td>&quot;all_properties&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;NODE(*)&quot;</td>
<td>&quot;architect&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;traverse&quot;</td>
<td>&quot;graph&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;NODE(*)&quot;</td>
<td>&quot;architect&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;read&quot;</td>
<td>&quot;all_properties&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;RELATIONSHIP(*)&quot;</td>
<td>&quot;architect&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;write&quot;</td>
<td>&quot;all_properties&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;RELATIONSHIP(*)&quot;</td>
<td>&quot;architect&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;traverse&quot;</td>
<td>&quot;graph&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;RELATIONSHIP(*)&quot;</td>
<td>&quot;architect&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;access&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;architect&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;schema&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;architect&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;token&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;architect&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;read&quot;</td>
<td>&quot;all_properties&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;NODE(*)&quot;</td>
<td>&quot;editor&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;write&quot;</td>
<td>&quot;all_properties&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;NODE(*)&quot;</td>
<td>&quot;editor&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;traverse&quot;</td>
<td>&quot;graph&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;NODE(*)&quot;</td>
<td>&quot;editor&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;read&quot;</td>
<td>&quot;all_properties&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;RELATIONSHIP(*)&quot;</td>
<td>&quot;editor&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;write&quot;</td>
<td>&quot;all_properties&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;RELATIONSHIP(*)&quot;</td>
<td>&quot;editor&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;traverse&quot;</td>
<td>&quot;graph&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;RELATIONSHIP(*)&quot;</td>
<td>&quot;editor&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;access&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;editor&quot;</td>
</tr>
<tr>
<td>&quot;DENIED&quot;</td>
<td>&quot;access&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;neo4j&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;noAccessUsers&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;read&quot;</td>
<td>&quot;all_properties&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;NODE(*)&quot;</td>
<td>&quot;publisher&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;write&quot;</td>
<td>&quot;all_properties&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;NODE(*)&quot;</td>
<td>&quot;publisher&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;traverse&quot;</td>
<td>&quot;graph&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;NODE(*)&quot;</td>
<td>&quot;publisher&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;read&quot;</td>
<td>&quot;all_properties&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;RELATIONSHIP(*)&quot;</td>
<td>&quot;publisher&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;write&quot;</td>
<td>&quot;all_properties&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;RELATIONSHIP(*)&quot;</td>
<td>&quot;publisher&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;traverse&quot;</td>
<td>&quot;graph&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;RELATIONSHIP(*)&quot;</td>
<td>&quot;publisher&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;access&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;publisher&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;token&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;publisher&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;token&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;publisher&quot;</td>
</tr>
</tbody>
</table>
Available privileges for a particular role can be seen using `SHOW ROLE name PRIVILEGES`.

**Query**

```
SHOW ROLE regularUsers PRIVILEGES
```

Lists all privileges for role 'regularUsers'

**Table 441. Result**

<table>
<thead>
<tr>
<th>access</th>
<th>action</th>
<th>resource</th>
<th>graph</th>
<th>segment</th>
<th>role</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;access&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;neo4j&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;regularUsers&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

Available privileges for a particular user can be seen using `SHOW USER name PRIVILEGES`.

**Query**

```
SHOW USER jake PRIVILEGES
```

Lists all privileges for user 'jake'

**Table 442. Result**

<table>
<thead>
<tr>
<th>access</th>
<th>action</th>
<th>resource</th>
<th>graph</th>
<th>segment</th>
<th>role</th>
<th>user</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;access&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;neo4j&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;regularUsers&quot;</td>
<td>&quot;jake&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

The **TRAVVERSE** privilege **Enterprise edition**

Users can be granted the right to find nodes and relationships using the `GRANT TRAVERSE` privilege.
Command syntax

GRANT TRAVERSE
  ON GRAPH[S] { * | name }
  [ ELEMENT[S] { * | label-or-rel-type[, ...] }
    NODE[S] { * | label[, ...] }
    RELATIONSHIP[S] { * | rel-type[, ...] }
  ]
  TO role[, ...]

For example, we can allow the user jake, who has role 'regularUsers' to find all nodes with the label Post.

Query

GRANT TRAVERSE ON GRAPH neo4j NODES Post TO regularUsers

0 rows, System updates: 1

The TRAVERSE privilege can also be denied.

Command syntax

DENY TRAVERSE
  ON GRAPH[S] { * | name }
  [ ELEMENT[S] { * | label-or-rel-type[, ...] }
    NODE[S] { * | label[, ...] }
    RELATIONSHIP[S] { * | rel-type[, ...] }
  ]
  TO role[, ...]

For example, we can disallow the user jake, who has role 'regularUsers' to find all nodes with the label Payments.

Query

DENY TRAVERSE ON GRAPH neo4j NODES Payments TO regularUsers

0 rows, System updates: 1

The READ privilege [Enterprise edition]

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 allowed to find in the first place.

Command syntax

GRANT READ
  "(* { * | property[, ...] })"
  ON GRAPH[S] { * | name }
  [ ELEMENT[S] { * | label-or-rel-type[, ...] }
    NODE[S] { * | label[, ...] }
    RELATIONSHIP[S] { * | rel-type[, ...] }
  ]
  TO role[, ...]
For example, we can allow the user jake, who has 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.

Query

| GRANT READ { * } ON GRAPH neo4j NODES Post TO regularUsers |

0 rows, System updates: 1

The READ privilege can also be denied.

Command syntax

| DENY READ { secret } ON GRAPH neo4j NODES Post TO regularUsers |

0 rows, System updates: 1

The MATCH privilege [Enterprise edition](#)

As a shorthand for TRAVERSE and READ, users can be granted the right to find and do property reads on nodes and relationships using the GRANT MATCH privilege.

Command syntax

| GRANT MATCH { * | property[ , ... ] } )" ON GRAPH[S] { * | name } |

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 a role regularUsers you can use the following GRANT MATCH query.
Like all other privileges, the MATCH privilege can also be denied.

**Command syntax**

```sql
DENY MATCH
  "({ * | property[, ...] })"
ON 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 a *. If you specify concrete property keys then DENY MATCH will only deny reading those properties. Finding the elements to traverse would still be allowed. If you specify * instead then both traversal of the element and all property reads will be disallowed. 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).

**Query**

```sql
DENY MATCH { content } ON GRAPH neo4j NODES Message TO regularUsers
```

0 rows, System updates: 1

The following query exemplifies how it would look like if you want to deny both reading all properties and traversing nodes labeled with Account.

**Query**

```sql
DENY MATCH { * } ON GRAPH neo4j NODES Account TO regularUsers
```

0 rows, System updates: 2

Please note that REVOKE MATCH is not allowed, instead revoke the individual READ and TRAVERSE privileges.

The **WRITE** privilege

The WRITE privilege enables you to write on a graph.
Command syntax

```sql
GRANT WRITE
ON GRAPH(S) { * | name }
TO role[, ...]
```

For example, granting the ability to write on the graph `neo4j` to the role `regularUsers` would be achieved using:

Query

```
GRANT WRITE ON GRAPH neo4j TO regularUsers
```

0 rows, System updates: 2

Unlike with `GRANT READ` it is not possible to restrict `WRITE` privileges to specific ELEMENTS, NODES or RELATIONSHIPS.

For example, using `NODES A` will produce a syntax error.

Query

```
GRANT WRITE ON GRAPH neo4j NODES A TO regularUsers
```

The use of ELEMENT, NODE or RELATIONSHIP with the WRITE privilege is not supported in this version.

```
(line 1, column 1 (offset: 0))
"GRANT WRITE ON GRAPH neo4j NODES A TO regularUsers"
```

The `WRITE` privilege can also be denied.

Command syntax

```sql
DENY WRITE
ON GRAPH(S) { * | name }
TO role[, ...]
```

For example, denying the ability to write on the graph `neo4j` to the role `regularUsers` would be achieved using:

Query

```
DENY WRITE ON GRAPH neo4j TO regularUsers
```

0 rows, System updates: 2

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 **REVOKE** command

Privileges that were granted or denied earlier can be revoked using the **REVOKE** command.

**Command syntax**

```
REVOKE
   [ GRANT | DENY ] privilege
   FROM role[, ...]
```

Please note that **REVOKE MATCH** is not allowed, instead revoke the individual **READ** and **TRAVERSE** privileges.

An example usage of the **REVOKE** command is given here:

**Query**

```
REVOKE GRANT TRAVERSE ON GRAPH neo4j NODES Post FROM regularUsers
```

0 rows, System updates: 1

While it can be explicitly specified that revoke should remove a **GRANT** or **DENY**, it is also possible to revoke either one by not specifying at all as the next example demonstrates. Because of this, if there happen to be a **GRANT** and a **DENY** on the same privilege, it would remove both.

**Query**

```
REVOKE TRAVERSE ON GRAPH neo4j NODES Payments FROM regularUsers
```

0 rows, System updates: 2

Some privileges are compound privileges and contains sub-privileges, for example **INDEX MANAGEMENT** which covers **CREATE INDEX** and **DROP INDEX**. When these compound privileges are revoked, all sub-privileges matching the revoke command will also be revoked as shown in the example below.

**Add CREATE INDEX and DROP INDEX privileges**

```
GRANT CREATE INDEX ON DATABASE * TO indexUsers
GRANT DROP INDEX ON DATABASE * TO indexUsers
```

**Query**

```
REVOKE INDEX MANAGEMENT ON DATABASE * FROM indexUsers
```

0 rows, System updates: 2

**Query**

```
SHOW ROLE indexUsers PRIVILEGES
```

Both the **CREATE INDEX** and **DROP INDEX** privileges have been revoked:
12.13.6. Security of administration

This section explains how to use Cypher to manage Neo4j administrative privileges.

All of the commands described in the enclosing Administration section require that the user executing the commands has the rights to do so. These privileges can be conferred either by granting the user the admin role, which enables all administrative rights, or by granting specific combinations of privileges.

- **The admin role**

- **Database administration**
  - The database ACCESS privilege
  - The database START/STOP privileges
  - The INDEX MANAGEMENT privileges
  - The CONSTRAINT MANAGEMENT privileges
  - The NAME MANAGEMENT privileges
  - Granting all database administration privileges

- **DBMS administration**
  - Using a custom role to manage DBMS privileges
  - The dbms ROLE MANAGEMENT privileges

The **admin role** [Enterprise edition](#)

The built-in role admin includes a number of privileges allowing users granted this role the ability to perform administrative tasks. These include the rights to perform the following classes of tasks:

- **Manage database security** for controlling 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 DBMS security** for controlling the rights to perform actions on the entire system:
  - Manage multiple databases
  - Manage users and roles
  - Change configuration parameters
  - Manage transactions
  - Manage sub-graph privileges
Manage procedure security

These rights are conferred using privileges that can be managed using `GRANT`, `DENY` and `REVOKE` commands, with the exception of the DBMS Security privileges which are only available within the built-in `admin` role.

Query

```
SHOW ROLE admin PRIVILEGES
```

Table 444. Result

<table>
<thead>
<tr>
<th>access</th>
<th>action</th>
<th>resource</th>
<th>graph</th>
<th>segment</th>
<th>role</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;read&quot;</td>
<td>&quot;all_properties&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;NODE(*)&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;write&quot;</td>
<td>&quot;all_properties&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;NODE(*)&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;traverse&quot;</td>
<td>&quot;graph&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;NODE(*)&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;read&quot;</td>
<td>&quot;all_properties&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;RELATIONSHIP(*)&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;write&quot;</td>
<td>&quot;all_properties&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;RELATIONSHIP(*)&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;traverse&quot;</td>
<td>&quot;graph&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;RELATIONSHIP(*)&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;access&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;admin&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;schema&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;token&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;admin&quot;</td>
</tr>
</tbody>
</table>

Rows: 10

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.

Database administration **Enterprise edition**

This section explains how to use Cypher to manage privileges for Neo4j database administrative rights.

As described in the section on sub-graph security, the `GRANT` command allows an administrator to grant a privilege to a role in order to access an entity. The `DENY` command allows an administrator to deny a privilege to a role in order to prevent access to an entity. The `REVOKE` command allows an administrator to remove a previously granted or denied privilege. The syntax is:

Table 445. General database privilege command syntax

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`GRANT database-privilege ON DATABASE[S] [*</td>
<td>name] TO role[,]`</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>DENY database-privilege ON DATABASE[S] {*</td>
<td>name} TO role[, ...]</td>
</tr>
<tr>
<td>REVOKE GRANT database-privilege ON DATABASE[S] {*</td>
<td>name} FROM role[, ...]</td>
</tr>
<tr>
<td>REVOKE DENY database-privilege ON DATABASE[S] {*</td>
<td>name} FROM role[, ...]</td>
</tr>
<tr>
<td>REVOKE database-privilege ON DATABASE[S] {*</td>
<td>name} FROM role[, ...]</td>
</tr>
</tbody>
</table>

Where the components are:

- **database-privilege**
  - **ACCESS**
    - allows access for a specific database/graph
  - **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.
  - **INDEX [MANAGEMENT]**
    - allows indexes to be created and deleted 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.
**CONSTRAINT [MANAGEMENT]**

allows constraints to be created and deleted on the specified database.

**CREATE NEW [NODE] LABEL**

allows labels to be created so that future nodes can be assigned them.

**CREATE NEW [RELATIONSHIP] TYPE**

allows relationship types to be created so that future relationships can be created with these types.

**CREATE NEW [PROPERTY] NAME**

allows property names to be created so that nodes and relationships can have properties with these names assigned.

**NAME [MANAGEMENT]**

allows all of the name management capabilities: node labels, relationship types and property names.

**ALL [[DATABASE] PRIVILEGES]**

allows access, start, stop, index, constraint, and name management for the specified database.

- **name**
  
  - The database to associate the privilege with. Note that if you delete a database and create a new one with the same name, the new one will NOT have any of the privileges specifically assigned to the deleted database.
  
  - It can be * which means all databases. Any new databases created after this command will also be associated with these privileges.

- **role[...]**

  - The role or roles to associate the privilege with, comma-separated.

**WARNING**

It is important to note that using DENY does NOT erase a GRANT command; they both exist. The only way to erase a privilege is with REVOKE.

The hierarchy between the different database privileges is shown in the image below.
Figure 3. Database privileges hierarchy

Table 446. Database privilege command syntax

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANT ACCESS ON DATABASE[S] [*</td>
<td>name} TO role[, ...]</td>
</tr>
<tr>
<td>GRANT {START</td>
<td>STOP} ON DATABASE[S] [*</td>
</tr>
<tr>
<td>GRANT {CREATE</td>
<td>DROP} INDEX[ES] ON DATABASE[S] [*</td>
</tr>
<tr>
<td>GRANT INDEX[ES] [MANAGEMENT] ON DATABASE[S] [*</td>
<td>name} TO role[, ...]</td>
</tr>
<tr>
<td>GRANT {CREATE</td>
<td>DROP} CONSTRAINT[S] ON DATABASE[S] [*</td>
</tr>
<tr>
<td>GRANT CONSTRAINT[S] [MANAGEMENT] ON DATABASE[S] [*</td>
<td>name} TO role[, ...]</td>
</tr>
<tr>
<td>GRANT CREATE NEW [NODE] LABEL[S] ON DATABASE[S] [*</td>
<td>name} TO role[, ...]</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>GRANT CREATE NEW [RELATIONSHIP] TYPE[S] ON DATABASE[S] {*</td>
<td>name} TO role[,...]</td>
</tr>
<tr>
<td>GRANT CREATE NEW [PROPERTY] NAME[S] ON DATABASE[S] {*</td>
<td>name} TO role[,...]</td>
</tr>
<tr>
<td>GRANT NAME [MANAGEMENT] ON DATABASE[S] {*</td>
<td>name} TO role[,...]</td>
</tr>
<tr>
<td>GRANT ALL [[DATABASE] PRIVILEGES] ON DATABASE[S] {*</td>
<td>name} TO role[,...]</td>
</tr>
</tbody>
</table>

Figure 4. Syntax of GRANT and DENY Database Privileges

The database **ACCESS** privilege

The **ACCESS** privilege enables users to connect to a database. With **ACCESS** you can run calculations, for example, `RETURN 2*5 AS answer` or call functions `RETURN timestamp() AS time`.

Command syntax

```
GRANT ACCESS ON DATABASE[S] {* | name} TO role[,...]
```

For example, granting the ability to access the database `neo4j` to the role `regularUsers` is done using the following query.
Query

```
GRANT ACCESS ON DATABASE neo4j TO regularUsers
```

0 rows, System updates: 1

The `ACCESS` privilege can also be denied.

Command syntax

```
DENY ACCESS ON DATABASE[s] [\* | name] TO role[, ...]
```

For example, denying the ability to access to the database `neo4j` to the role `regularUsers` is done using the following query.

Query

```
DENY ACCESS ON DATABASE neo4j TO regularUsers
```

0 rows, System updates: 1

The privileges granted can be seen using the `SHOW PRIVILEGES` command:

Query

```
SHOW ROLE regularUsers PRIVILEGES
```

<table>
<thead>
<tr>
<th>access</th>
<th>action</th>
<th>resource</th>
<th>graph</th>
<th>segment</th>
<th>role</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;DENIED&quot;</td>
<td>&quot;access&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;neo4j&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;regularUsers&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;access&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;neo4j&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;regularUsers&quot;</td>
</tr>
</tbody>
</table>

Rows: 2

Only the `admin` role has access to execute the `SHOW PRIVILEGES` command.

The database `START/STOP` privileges\(^\text{Enterprise edition}\)

The `START` privilege can be used to enable the ability to start a database.

Command syntax

```
GRANT START ON DATABASE[s] [\* | name] TO role[, ...]
```

For example, granting the ability to start the database `neo4j` to the role `regularUsers` is done using the following query.
Query

GRANT START ON DATABASE neo4j TO regularUsers

0 rows, System updates: 1

The START privilege can also be denied.

Command syntax

DENY START ON DATABASE [S] {* | name} TO role[,...]

For example, denying the ability to start to the database neo4j to the role regularUsers is done using the following query.

Query

DENY START ON DATABASE system TO regularUsers

0 rows, System updates: 1

The STOP privilege can be used to enable the ability to stop a database.

Command syntax

GRANT STOP ON DATABASE [S] {* | name} TO role[,...]

For example, granting the ability to stop the database neo4j to the role regularUsers is done using the following query.

Query

GRANT STOP ON DATABASE neo4j TO regularUsers

0 rows, System updates: 1

The STOP privilege can also be denied.

Command syntax

DENY STOP ON DATABASE [S] {* | name} TO role[,...]

For example, denying the ability to stop to the database neo4j to the role regularUsers is done using the following query.
DENY STOP ON DATABASE system TO regularUsers

0 rows, System updates: 1

The privileges granted can be seen using the `SHOW PRIVILEGES` command:

```
SHOW ROLE regularUsers PRIVILEGES
```

<table>
<thead>
<tr>
<th>access</th>
<th>action</th>
<th>resource</th>
<th>graph</th>
<th>segment</th>
<th>role</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;DENIED&quot;</td>
<td>&quot;access&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;neo4j&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;regularUsers&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;access&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;neo4j&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;regularUsers&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;start_database&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;neo4j&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;regularUsers&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;stop_database&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;neo4j&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;regularUsers&quot;</td>
</tr>
<tr>
<td>&quot;DENIED&quot;</td>
<td>&quot;start_database&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;system&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;regularUsers&quot;</td>
</tr>
<tr>
<td>&quot;DENIED&quot;</td>
<td>&quot;stop_database&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;system&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;regularUsers&quot;</td>
</tr>
</tbody>
</table>

Rows: 6

The `INDEX MANAGEMENT` privileges [Enterprise edition](#)

Indexes can be created or deleted with the `CREATE INDEX` and `DROP INDEX` commands. The privilege to do this can be granted with `GRANT CREATE INDEX` and `GRANT DROP INDEX` commands.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANT {CREATE</td>
<td>DROP} INDEX[ES] ON DATABASE[S] {*</td>
</tr>
<tr>
<td>GRANT INDEX[ES] [MANAGEMENT] ON DATABASE[S] {*</td>
<td>name} TO role[, ...]</td>
</tr>
</tbody>
</table>

For example, granting the ability to create indexes on the database `neo4j` to the role `regularUsers` is done using the following query.

```
GRANT CREATE INDEX ON DATABASE neo4j TO regularUsers
```
The **CONSTRAINT MANAGEMENT** privileges [Enterprise edition](#)

Constraints can be created or deleted with the `CREATE CONSTRAINT` and `DROP CONSTRAINT` commands. The privilege to do this can be granted with `GRANT CREATE CONSTRAINT` and `GRANT DROP CONSTRAINT` commands.

**Table 450. Constraint management command syntax**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`GRANT {CREATE</td>
<td>DROP} CONSTRAINT[S] ON DATABASE[S] {*</td>
</tr>
<tr>
<td>`GRANT CONSTRAINT[S] [MANAGEMENT] ON DATABASE[S] {*</td>
<td>name} TO role[, ...]`</td>
</tr>
</tbody>
</table>

For example, granting the ability to create constraints on the database `neo4j` to the role `regularUsers` is done using the following query.

**Query**

```
GRANT CREATE CONSTRAINT ON DATABASE neo4j TO regularUsers
```

0 rows, System updates: 1

The **NAME MANAGEMENT** privileges [Enterprise edition](#)

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 451. Label, relationship type and property name management command syntax**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`GRANT CREATE NEW [NODE] LABEL[S] ON DATABASE[S] {*</td>
<td>name} TO role[, ...]`</td>
</tr>
<tr>
<td>`GRANT CREATE NEW [RELATIONSHIP] TYPE[S] ON DATABASE[S] {*</td>
<td>name} TO role[, ...]`</td>
</tr>
</tbody>
</table>
For example, granting the ability to create new properties on nodes or relationships in the database `neo4j` to the role `regularUsers` is done using the following query.

Query

```
GRANT CREATE NEW PROPERTY NAME ON DATABASE neo4j TO regularUsers
```

0 rows, System updates: 1

Granting all database administration privileges [Enterprise edition](#)

Conferring the right to access, start, and stop, as well as perform index, constraint, and name management can be achieved with a single command:

Command syntax

```
GRANT ALL DATABASE PRIVILEGES ON DATABASE neo4j TO regularUsers
```

For example, granting the ability to access, start and stop all databases and create indexes, constraints, labels, relationship types and property names on the database `neo4j` to the role `regularUsers` is done using the following query.

Query

```
GRANT ALL DATABASE PRIVILEGES ON DATABASE neo4j TO regularUsers
```

0 rows, System updates: 4

The privileges granted can be seen using the `SHOW PRIVILEGES` command:

Query

```
SHOW ROLE regularUsers PRIVILEGES
```

Table 452. Result
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.

As described above, the admin role has a number of built-in privileges that cannot be assigned using Cypher commands. These include:

- Create or drop databases
- Change configuration parameters
- Manage transactions
- Manage users and roles (role management by itself is assignable using Cypher commands)
- Manage sub-graph privileges
- Manage procedure security

The easiest way to enable a user to perform these tasks is to grant them the admin role. The only subset of these privileges that is assignable using Cypher commands is role management. However, it is possible to make a custom role with a subset of these privileges.

Using a custom role to manage DBMS privileges

If it is desired to have an administrator with a subset of privileges that includes all DBMS privileges, but not all database privileges, this can be achieved by copying the admin role and revoking or denying some
privileges.

First we copy the 'admin' role:

Query

```
CREATE ROLE usermanager AS COPY OF admin
```

0 rows, System updates: 2

Then we DENY ACCESS to normal databases:

Query

```
DENY ACCESS ON DATABASE * TO usermanager
```

0 rows, System updates: 1

And DENY START and STOP for normal databases:

Query

```
DENY START ON DATABASE * TO usermanager
```

0 rows, System updates: 1

Query

```
DENY STOP ON DATABASE * TO usermanager
```

0 rows, System updates: 1

And DENY index and constraint management:

Query

```
DENY INDEX MANAGEMENT ON DATABASE * TO usermanager
```

0 rows, System updates: 2

Query

```
DENY CONSTRAINT MANAGEMENT ON DATABASE * TO usermanager
```

0 rows, System updates: 2

And finally DENY label, relationship type and property name:

Query

```
DENY NAME MANAGEMENT ON DATABASE * TO usermanager
```
The resulting role should have privileges that only allow the DBMS capabilities, like user and role management:

Query

```
SHOW ROLE usermanager PRIVILEGES
```

Lists all privileges for role 'usermanager'

Table 453. Result

<table>
<thead>
<tr>
<th>access</th>
<th>action</th>
<th>resource</th>
<th>graph</th>
<th>segment</th>
<th>role</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;read&quot;</td>
<td>&quot;all_properties&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;NODE(*)&quot;</td>
<td>&quot;usermanager&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;write&quot;</td>
<td>&quot;all_properties&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;NODE(*)&quot;</td>
<td>&quot;usermanager&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;traverse&quot;</td>
<td>&quot;graph&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;NODE(*)&quot;</td>
<td>&quot;usermanager&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;read&quot;</td>
<td>&quot;all_properties&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;RELATIONSHIP(*)&quot;</td>
<td>&quot;usermanager&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;write&quot;</td>
<td>&quot;all_properties&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;RELATIONSHIP(*)&quot;</td>
<td>&quot;usermanager&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;traverse&quot;</td>
<td>&quot;graph&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;RELATIONSHIP(*)&quot;</td>
<td>&quot;usermanager&quot;</td>
</tr>
<tr>
<td>&quot;DENIED&quot;</td>
<td>&quot;access&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;usermanager&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;access&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;usermanager&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;admin&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;usermanager&quot;</td>
</tr>
<tr>
<td>&quot;DENIED&quot;</td>
<td>&quot;create_constraint&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;usermanager&quot;</td>
</tr>
<tr>
<td>&quot;DENIED&quot;</td>
<td>&quot;create_index&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;usermanager&quot;</td>
</tr>
<tr>
<td>&quot;DENIED&quot;</td>
<td>&quot;create_label&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;usermanager&quot;</td>
</tr>
<tr>
<td>&quot;DENIED&quot;</td>
<td>&quot;create_propertykey&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;usermanager&quot;</td>
</tr>
<tr>
<td>&quot;DENIED&quot;</td>
<td>&quot;create_reltype&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;usermanager&quot;</td>
</tr>
<tr>
<td>&quot;DENIED&quot;</td>
<td>&quot;drop_constraint&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;usermanager&quot;</td>
</tr>
<tr>
<td>&quot;DENIED&quot;</td>
<td>&quot;drop_index&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;usermanager&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;schema&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;usermanager&quot;</td>
</tr>
<tr>
<td>&quot;DENIED&quot;</td>
<td>&quot;start_database&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;usermanager&quot;</td>
</tr>
<tr>
<td>&quot;DENIED&quot;</td>
<td>&quot;stop_database&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;usermanager&quot;</td>
</tr>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;token&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;usermanager&quot;</td>
</tr>
</tbody>
</table>

Rows: 20

The dbms ROLE MANAGEMENT privileges [Enterprise edition](#)

The dbms privileges for role management are assignable using Cypher administrative commands. They can be granted, denied and revoked like other privileges.
Figure 5. Role management privileges hierarchy

Table 454. Role management privileges command syntax

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANT CREATE ROLE</td>
<td>Enable the specified role or roles to create new roles</td>
</tr>
<tr>
<td>ON DBMS</td>
<td></td>
</tr>
<tr>
<td>TO role[, ...]</td>
<td></td>
</tr>
<tr>
<td>GRANT DROP ROLE</td>
<td>Enable the specified role or roles to delete roles</td>
</tr>
<tr>
<td>ON DBMS</td>
<td></td>
</tr>
<tr>
<td>TO role[, ...]</td>
<td></td>
</tr>
<tr>
<td>GRANT ASSIGN ROLE</td>
<td>Enable the specified role or roles to assign roles to users</td>
</tr>
<tr>
<td>ON DBMS</td>
<td></td>
</tr>
<tr>
<td>TO role[, ...]</td>
<td></td>
</tr>
<tr>
<td>GRANT REMOVE ROLE</td>
<td>Enable the specified role or roles to remove roles from users</td>
</tr>
<tr>
<td>ON DBMS</td>
<td></td>
</tr>
<tr>
<td>TO role[, ...]</td>
<td></td>
</tr>
<tr>
<td>GRANT SHOW ROLE</td>
<td>Enable the specified role or roles to list roles</td>
</tr>
<tr>
<td>ON DBMS</td>
<td></td>
</tr>
<tr>
<td>TO role[, ...]</td>
<td></td>
</tr>
<tr>
<td>GRANT ROLE MANAGEMENT</td>
<td>Enable the specified role or roles to create, delete, assign, remove and</td>
</tr>
<tr>
<td>ON DBMS</td>
<td>list roles</td>
</tr>
<tr>
<td>TO role[, ...]</td>
<td></td>
</tr>
</tbody>
</table>

Only the admin role has access to execute the SHOW PRIVILEGES command.

The ability to add roles can be granted via the CREATE ROLE privilege. The following query shows an example of this:
Query

GRANT CREATE ROLE ON DBMS TO roleAdder

0 rows, System updates: 1

The resulting role should have privileges that only allow adding roles:

Query

SHOW ROLE roleAdder PRIVILEGES

Lists all privileges for role 'roleAdder'

Table 455. Result

<table>
<thead>
<tr>
<th>access</th>
<th>action</th>
<th>resource</th>
<th>graph</th>
<th>segment</th>
<th>role</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;create_role&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;roleAdder&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

The ability to delete roles can be granted via the DROP ROLE privilege. The following query shows an example of this:

Query

GRANT DROP ROLE ON DBMS TO roleDropper

0 rows, System updates: 1

The resulting role should have privileges that only allow deleting roles:

Query

SHOW ROLE roleDropper PRIVILEGES

Lists all privileges for role 'roleDropper'

Table 456. Result

<table>
<thead>
<tr>
<th>access</th>
<th>action</th>
<th>resource</th>
<th>graph</th>
<th>segment</th>
<th>role</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;drop_role&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;roleDropper&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

The ability to assign roles to users can be granted via the ASSIGN ROLE privilege. The following query shows an example of this:

Query

GRANT ASSIGN ROLE ON DBMS TO roleAssigner
The resulting role should have privileges that only allow assigning/granting roles:

Query

SHOW ROLE roleAssigner PRIVILEGES

Lists all privileges for role 'roleAssigner'

Table 457. Result

<table>
<thead>
<tr>
<th>access</th>
<th>action</th>
<th>resource</th>
<th>graph</th>
<th>segment</th>
<th>role</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;assign_role&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;roleAssigner&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

The ability to remove roles from users can be granted via the REMOVE ROLE privilege. The following query shows an example of this:

Query

GRANT REMOVE ROLE ON DBMS TO roleRemover

0 rows, System updates: 1

The resulting role should have privileges that only allow removing/revoking roles:

Query

SHOW ROLE roleRemover PRIVILEGES

Lists all privileges for role 'roleRemover'

Table 458. Result

<table>
<thead>
<tr>
<th>access</th>
<th>action</th>
<th>resource</th>
<th>graph</th>
<th>segment</th>
<th>role</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;remove_role&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;roleRemover&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

The ability to show roles can be granted via the SHOW ROLE privilege. The following query shows an example of this:

Query

GRANT SHOW ROLE ON DBMS TO roleShower

0 rows, System updates: 1

The resulting role should have privileges that only allow showing roles, not privileges:
Query

```
SHOW ROLE roleShower PRIVILEGES
```

Lists all privileges for role 'roleShower'

Table 459. Result

<table>
<thead>
<tr>
<th>access</th>
<th>action</th>
<th>resource</th>
<th>graph</th>
<th>segment</th>
<th>role</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;show_role&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;roleShower&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

The privileges to create, delete, assign, remove, and list roles can be granted via the ROLE MANAGEMENT privilege. The following query shows an example of this:

Query

```
GRANT ROLE MANAGEMENT ON DBMS TO roleManager
```

0 rows, System updates: 1

The resulting role should have all privileges to manage roles:

Query

```
SHOW ROLE roleManager PRIVILEGES
```

Lists all privileges for role 'roleManager'

Table 460. Result

<table>
<thead>
<tr>
<th>access</th>
<th>action</th>
<th>resource</th>
<th>graph</th>
<th>segment</th>
<th>role</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GRANTED&quot;</td>
<td>&quot;role_management&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;*&quot;</td>
<td>&quot;database&quot;</td>
<td>&quot;roleManager&quot;</td>
</tr>
</tbody>
</table>

Rows: 1

12.13.7. Known limitations of security

This section explains known limitations and implications of Neo4j's role-based access control security.

Security and indexes

As described in Indexes for search performance, Neo4j 4.0 supports the creation and use of indexes to improve the performance of Cypher queries. The Neo4j security model will impact the results of queries (regardless if the indexes are used). 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 occurred for each specific entry returned from the index. As a result, Neo4j will return zero results from full-text indexes if it is determined that any result might violate the security privileges active for that query.

Since full-text indexes are not automatically used by Cypher, this does not lead to the case where the same Cypher query would return different results simply because such an index got created. Users need to explicitly call procedures to use these indexes. The problem is only that if this behavior is not understood 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 these have properties name and surname. We have indexes on both properties:

![Code Block]

```
CREATE INDEX singleProp FOR (n:User) ON (n.name);
CREATE INDEX composite FOR (n:User) ON (n.name, n.surname);
CALL db.index.fulltext.createNodeIndex("userNames","["User", "Person"],["name", "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 fulltext 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 the index.

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, we 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:

- do a scan on the index to create a stream of results of nodes with the name property, which leads to five results
- filter the results to include only nodes where n.name CONTAINS 'ndy', filtering out Mark and Joe so we have three results
- filter the results to exclude nodes that also have the denied label :Person, filtering out Mandy so we have two results

For the above dataset, we can see we will get two results and that only one of these has the surname property.

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, almost identical to the single property index query:

- do a scan on the index to create a stream of results of nodes with the name and surname property, which leads to four results
- filter the results to include only nodes where n.name CONTAINS 'ndy', filtering out Mark and Joe so we have two results
- filter the results to exclude nodes that also have the denied label :Person, filtering out Mandy so we only have one result

For the above dataset, we can see we will get one result.

What if we query this with the full-text index:

```
CALL db.index.fulltext.queryNodes("userNames", "ndy") YIELD node, score
RETURN node.name
```

The problem now is that we do not know if the results provided by the index were because of 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 so we have three results.

This difference in results is due to 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;

Now we 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 this query relates to the denied property.

However, for the query targeting the composite index, things have changed.

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, we do not know if the results provided by the index were because of a match on the name or the surname property. Results from the surname now need to be excluded by the security rules, because they require that the user cannot see any surname properties. However, the security model is not able to introspect the Lucene query to know what it will actually do, whether it works only on the allowed name property, or also on the disallowed surname property. We know that the earlier query returned a match for Joe Andy which should now be filtered out. So, in order to never return results the user should not be able to see, we have to block all results. 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 process as before

The query will therefore return zero results in this case, rather than simply returning the results Andy and Sandy which might be 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 because of their ability to have multiple labels. Relationships can only ever have one type 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 labels attached to the node by calling the built-in `labels()` function. In the case of nodes with multiple labels, this can seemingly result in labels being returned to which the user wasn’t directly granted access to.

To give an illustrative example, imagine a graph with three nodes: one labeled :A, one labeled :B and one with :A :B. We also have a user with a role `custom` as 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 be returned 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);
```

will return only the one node that has both labels: :A :B. Even though :B was not allowed access for traversal, there is one node with that label accessible in the data because of the allowlisted label :A that is attached to the same node.

If a user is denied 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. For our 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.
To reuse the example of the previous section: imagine a graph with three nodes: one labeled :A, one labeled :B and one with :A :B. We also have a user with a role custom as defined by:

```
GRANT TRAVERSE ON GRAPH * NODES A TO custom;
```

This means that only label :A is explicitly allowlisted. 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 comes down to necessary additional security checks that incur additional data accesses. Especially in regards to count store operations, as they are usually very fast lookups, the difference might be noticeable.

Let's look at 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.
<table>
<thead>
<tr>
<th>User with <strong>free</strong> role</th>
<th>User with <strong>restricted</strong> role</th>
</tr>
</thead>
<tbody>
<tr>
<td>The database can access the count store and retrieve the total number of nodes with the label :Person. This is a very quick operation.</td>
<td>The database cannot just access the count store because it must make sure that only traversable nodes with the desired label :Person are counted. Due to this, each node with the :Person label needs to be accessed and examined to make sure that it does not also have a denylisted label, such as :Customer. Due to the additional data accesses that the security checks need to do, this operation will be slower compared to executing the query as an unrestricted user.</td>
</tr>
</tbody>
</table>
Chapter 13. Query tuning

This section describes query tuning for the Cypher query language.

Neo4j aims to execute queries as fast as possible.

However, when optimizing for maximum query execution performance, it may be helpful to rephrase queries using knowledge about the domain and the application.

The overall goal of manual query performance optimization is to ensure that only necessary data is retrieved from the graph. At the very least, data should get filtered out as early as possible in order to reduce the amount of work that has to be done in the later stages of query execution. This also applies to what gets returned: returning whole nodes and relationships ought to be avoided in favour of selecting and returning only the data that is needed. You should also make sure to set an upper limit on variable length patterns, so they don’t cover larger portions of the dataset than needed.

Each Cypher query gets optimized and transformed into an execution plan by the Cypher query planner. To minimize the resources used for this, try to use parameters instead of literals when possible. This allows Cypher to re-use your queries instead of having to parse and build new execution plans.

To read more about the execution plan operators mentioned in this chapter, see Execution plans.

- Cypher query options
- Profiling a query
- The use of indexes
- Basic query tuning example
- Advanced query tuning example
  - Introduction
  - The data set
  - Index-backed property-lookup
  - Index-backed order by
- Planner hints and the USING keyword
  - Introduction
  - Index hints
  - Scan hints
  - Join hints
  - PERIODIC COMMIT query hint

13.1. Cypher query options

This section describes the query options available in Cypher.
Query execution can be fine-tuned through the use of query options. In order to use one or more of these options, the query must be prepended with `CYPHER`, followed by the query option(s), as exemplified thus:

```
CYPHER query-option [further-query-options] query.
```

### 13.1.1. Cypher version

Occasionally, there is a requirement to use a previous version of the Cypher compiler when running a query. Here we detail the available versions:

<table>
<thead>
<tr>
<th>Query option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5</td>
<td>This will force the query to use Neo4j Cypher 3.5.</td>
<td></td>
</tr>
<tr>
<td>4.0</td>
<td>This will force the query to use Neo4j Cypher 4.0. As this is the default version, it is not necessary to use this option explicitly.</td>
<td>X</td>
</tr>
</tbody>
</table>

In Neo4j 4.0, the support for Cypher 3.5 is provided only at the parser level. The consequence is that some underlying features available in Neo4j 3.5 are no longer available and will result in runtime errors.

Please refer to the discussion in Cypher Compatibility for more information on which features are affected.

### 13.1.2. Cypher runtime

Using the execution plan, the query is executed — and records returned — by the Cypher runtime. Depending on whether Neo4j Enterprise Edition or Neo4j Community Edition is used, there are three different runtimes available:

**Interpreted**

In this runtime, the operators in the execution plan are chained together in a tree, where each non-leaf operator feeds from one or two child operators. The tree thus comprises nested iterators, and the records are streamed in a pipelined manner from the top iterator, which pulls from the next iterator and so on.

**Slotted**

This is very similar to the interpreted runtime, except that there are additional optimizations regarding the way in which the records are streamed through the iterators. This results in improvements to both the performance and memory usage of the query. In effect, this can be thought of as the ‘faster interpreted’ runtime.

**Pipelined**

The pipelined runtime was introduced in Neo4j 4.0 as a replacement for the older compiled runtime used in the Neo4j 3.x versions. It combines some of the advantages of the compiled runtime in a new architecture that allows for support of a wider range of queries.
Algorithms are employed to intelligently group the operators in the execution plan in order to generate new combinations and orders of execution which are optimised for performance and memory usage. While this should lead to superior performance in most cases (over both the interpreted and slotted runtimes), it is still under development and does not support all possible operators or queries (the slotted runtime covers all operators and queries).

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>runtime=interpreted</td>
<td>This will force the query planner to use the interpreted runtime.</td>
<td>This is not used in Enterprise Edition unless explicitly asked for. It is the only option for all queries in Community Edition—it is not necessary to specify this option in Community Edition.</td>
</tr>
<tr>
<td>runtime=slotted</td>
<td>This will cause the query planner to use the slotted runtime.</td>
<td>This is the default option for all queries which are not supported by runtime=pipelined in Enterprise Edition.</td>
</tr>
<tr>
<td>runtime=pipelined</td>
<td>This will cause the query planner to use the pipelined runtime if it supports the query. If the pipelined runtime does not support the query, the planner will fall back to the slotted runtime.</td>
<td>This is the default option for some queries in Enterprise Edition.</td>
</tr>
</tbody>
</table>

In Enterprise Edition, the Cypher query planner selects the runtime, falling back to alternative runtimes as follows:

- Try the pipelined runtime first.
- If the pipelined runtime does not support the query, then fall back to use the slotted runtime.
- Finally, if the slotted runtime does not support the query, fall back to the interpreted runtime. The interpreted runtime supports all queries, and is the only option in Neo4j Community Edition.

### 13.2. Profiling a query

There are two options to choose from when you want to analyze a query by looking at its execution plan:

**EXPLAIN**

If you want to see the execution plan but not run the statement, prepend your Cypher statement with **EXPLAIN**. The statement will always return an empty result and make no changes to the database.

**PROFILE**

If you want to run the statement and see which operators are doing most of the work, use **PROFILE**. This
will run your statement and keep track of how many rows pass through each operator, and how much each operator needs to interact with the storage layer to retrieve the necessary data. Please note that profiling your query uses more resources, so you should not profile unless you are actively working on a query.

See Execution plans for a detailed explanation of each of the operators contained in an execution plan.

|💡 | Being explicit about what types and labels you expect relationships and nodes to have in your query helps Neo4j use the best possible statistical information, which leads to better execution plans. This means that when you know that a relationship can only be of a certain type, you should add that to the query. The same goes for labels, where declaring labels on both the start and end nodes of a relationship helps Neo4j find the best way to execute the statement. |

13.3. The use of indexes

This section describes the query plans when indexes are used in various scenarios.

The task of tuning calls for different indexes depending on what the queries look like. Therefore, it is important to have a fundamental understanding of how the indexes operate. This section describes the query plans that result from different index scenarios.

Please refer to Indexes for search performance for instructions on how to create and maintain the indexes themselves.

13.3.1. A simple example

In the example below, the query will use a Person(firstname) index, if it exists.

Query

```
MATCH (person:Person {firstname: 'Andy'}) RETURN person
```
13.3.2. Equality check using **WHERE** (single-property index)

A query containing equality comparisons of a single indexed property in the **WHERE** clause is backed automatically by the index. It is also possible for a query with multiple OR predicates to use multiple indexes, if indexes exist on the properties. For example, if indexes exist on both **:Label(p1)** and **:Label(p2)**, MATCH (n:Label) WHERE n.p1 = 1 OR n.p2 = 2 RETURN n will use both indexes.

**Query**

```
MATCH (person:Person) WHERE person.firstname = 'Andy' RETURN person
```

**Query Plan**

```
Compiler CYPHER 4.0
Planner COST
Runtime PIPELINED
Runtime version 4.0

| Operator        | Estimated Rows | Rows | DB Hits | Order                | Variables | Other |
|-----------------+----------------+------+---------+----------------------|-----------+-------|
| +ProduceResults |              1 |    1 |       0 | person.firstname ASC | person    |       |
| |               +----------------+------+---------+----------------------|-----------+-------|
| +NodeIndexSeek  |              1 |    1 |       2 | person.firstname ASC | person    |       |
| :Person(firstname) |                   |       |         |                      |           |
| +-----------------+----------------+------+---------+----------------------|-----------+-------|

Total database accesses: 2
```
13.3.3. Equality check using **WHERE** (composite index)

A query containing equality comparisons for all the properties of a composite index will automatically be backed by the same index. However, the query does not need to have equality on all properties. It can have ranges and existence predicates as well. But in these cases rewrites might happen depending on which properties have which predicates, see **composite index limitations**. The following query will use the composite index defined earlier:

Query

```
MATCH (n:Person) WHERE n.age = 35 AND n.country = 'UK' RETURN n
```

However, the query `MATCH (n:Person) WHERE n.age = 35 RETURN n` will not be backed by the composite index, as the query does not contain a predicate on the country property. It will only be backed by an index on the Person label and age property defined thus: `:Person(age);` i.e. a single-property index.

Result

```
+----------------------------------------------------------+
| n                                                                                         |
+----------------------------------------------------------+
| Node[0]{country:"UK",highScore:54321,firstname:"John",surname:"Smith",name:"john",age:35} |
+----------------------------------------------------------+
```

```
1 row
```

13.3.4. 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 (person:Person) WHERE person.firstname > 'B' RETURN person
```
13.3.5. 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 (person:Person) WHERE person.firstname > 'B' AND person.highScore > 10000 RETURN person
```
13.3.6. 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
```
13.3.7. 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 exists(person.name) RETURN person
```

**Query Plan**

```
Compiler CYPHER 4.0
Planner COST
Runtime PIPELINED
Runtime version 4.0

| Operator                      | Estimated Rows | Rows | DB Hits | Time (ms) | Variables | Other
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.133</td>
<td>person</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+NodeIndexSeek(range,exists)</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>6.807</td>
<td>person</td>
</tr>
<tr>
<td>:Person(highScore,name)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Total database accesses: 2
13.3.8. List membership check using **IN** (single-property index)

The **IN** predicate on `person.firstname` in the following query will use the single-property index `Person(firstname)` if it exists.

Query

```
MATCH (person:Person) WHERE person.firstname IN ['Andy', 'John'] RETURN person
```

Query Plan

```
Compiler CYPHER 4.0
Planner COST
Runtime PIPELINED
Runtime version 4.0
+-----------------+----------------+------+---------+-----------+--------------------+
| Operator        | Estimated Rows | Rows | DB Hits | Variables | Other              |
| +-----------------+----------------+------+---------+-----------+--------------------+
| +ProduceResults  | 24 | 2 | 0 | person | 
| | +-----------------+----------------+------+---------+-----------+--------------------+
| +NodeIndexSeek   | 24 | 2 | 0 | person | :Person(firstname) | 
| +-----------------+----------------+------+---------+-----------+--------------------+
Total database accesses: 0
```

13.3.9. List membership check using **IN** (composite index)

The **IN** predicates on `person.age` and `person.country` in the following query will use the composite index `Person(age, country)` if it exists.

Query

```
MATCH (person:Person) WHERE person.age IN [10, 20, 35] AND person.country IN ['Sweden', 'USA', 'UK'] RETURN person
```

```
13.3.10. Prefix search using **STARTS WITH** (single-property index)

The **STARTS WITH** predicate on `person.firstname` in the following query will use the `Person(firstname)` index, if it exists.

**Query**

```
MATCH (person:Person) WHERE person.firstname STARTS WITH 'And' RETURN person
```

**Query Plan**

```
Compiler CYPHER 4.0
Planner COST
Runtime PIPELINED
Runtime version 4.0

+-----------------------+----------------+------+---------+-----------+----------------------+-----------+
| Operator              | Estimated Rows | Rows | DB Hits | Time (ms) | Order                | Variables |
| Other                |                |      |         |           |                     |           |
| +ProduceResults       |              2 |    1 |       0 |     0.106 | person.firstname ASC | person    |
| |                     +----------------+------+---------+-----------+----------------------+-----------+
| +NodeIndexSeekByRange |              2 |    1 |       2 |     0.206 | person.firstname ASC | person    |
| :Person(firstname STARTS WITH $` AUTOSTRING0`) |                |      |         |           |                     |           |
| +-------------------------------+----------------+------+---------+-----------+----------------------+-----------+----------------------+-----------+
Total database accesses: 2
```
13.3.11. 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 exists(person.surname) RETURN person
```

**Query Plan**

- Compiler CYPHER 4.0
- Planner COST
- Runtime PIPELINED
- Runtime version 4.0

<table>
<thead>
<tr>
<th>Operator</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Time (ms)</th>
<th>Variables</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0.100</td>
<td>person</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>0.100</td>
<td>person</td>
</tr>
<tr>
<td>+NodeIndexSeek(range,exists)</td>
<td></td>
<td>1</td>
<td>2</td>
<td>0.257</td>
<td>person</td>
<td></td>
</tr>
<tr>
<td>:Person(firstname,surname)</td>
<td></td>
<td>1</td>
<td>2</td>
<td>0.257</td>
<td>person</td>
<td></td>
</tr>
</tbody>
</table>

Total database accesses: 2

13.3.12. Suffix search using **ENDS WITH** (single-property index)

The **ENDS WITH** predicate on `person.firstname` in the following query will use the `Person(firstname)` index, if it exists. All values stored in the `Person(firstname)` index will be searched, and entries ending with 'hn' will be returned. This means that although the search will not be optimized to the extent of queries using =, **IN**, >, < or **STARTS WITH**, it is still faster than not using an index in the first place. Composite indexes are currently not able to support **ENDS WITH**.

**Query**

```
MATCH (person:Person) WHERE person.firstname ENDS WITH 'hn' RETURN person
```
### 13.3.13. Suffix search using **ENDS WITH** (composite index)

The **ENDS WITH** predicate on `person.surname` in the following query will use the `Person(surname,age)` index, if it exists. However, it will be rewritten as existence check and a filter due to the index not supporting actual suffix searches for composite indexes, this is still faster than not using an index in the first place.

Any (non-existence check) predicate on `person.age` will also be rewritten as existence check with a filter.

More information about how the rewriting works can be found in [composite index limitations](#).

**Query**

```cypher
MATCH (person:Person) WHERE person.surname ENDS WITH '300' AND exists(person.age) RETURN person
```
### 13.3.14. Substring search using CONTAINS (single-property index)

The **CONTAINS** predicate on `person.firstname` in the following query will use the `Person(firstname)` index, if it exists. All values stored in the `Person(firstname)` index will be searched, and entries containing 'h' will be returned. This means that although the search will not be optimized to the extent of queries using `=`, `IN`, `>`, `<` or `STARTS WITH`, it is still faster than not using an index in the first place. Composite indexes are currently not able to support **CONTAINS**.

**Query**

```
MATCH (person:Person) WHERE person.firstname CONTAINS 'h' RETURN person
```
13.3.15. Substring search using **CONTAINS** (composite index)

The **CONTAINS** predicate on `person.surname` in the following query will use the `Person(surname,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.surname CONTAINS '300' AND exists(person.age) RETURN person
```
13.3.16. Existence check using `exists` (single-property index)

The `exists(p.firstname)` predicate in the following query will use the `Person(firstname)` index, if it exists.

Query

```cypher
MATCH (p:Person) WHERE exists(p.firstname) RETURN p
```

13.3.17. Existence check using `exists` (composite index)

The `exists(p.firstname)` and `exists(p.surname)` predicate 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.
MATCH (p:Person) WHERE exists(p.firstname) AND exists(p.surname) RETURN p

13.3.18. 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.

MATCH (p:Person) WHERE distance(p.location, point({x: 1, y: 2})) < 2 RETURN p.location
13.3.19. Spatial distance searches (composite index)

If a property with point values is indexed, the index is used for spatial distance searches as well as for range queries. Any following (non-existence check) predicates (here on property p.name for index :Person(place, name)) will be rewritten as existence check with a filter.

Query

```
MATCH (p:Person) WHERE distance(p.place, point({x: 1, y: 2})) < 2 AND exists(p.name) RETURN p.place
```
13.3.20. Spatial bounding box searches (single-property index)

The ability to do index seeks on bounded ranges works even with the 2D and 3D spatial \textit{Point} types.

\textbf{Query}

\begin{verbatim}
MATCH (person:Person) WHERE point({x: $\text{AUTOINT0}$, y: $\text{AUTOINT1}$}) < person.location < point({x: 2, y: 6}) RETURN person
\end{verbatim}
13.3.21. Spatial bounding box searches (composite index)

The ability to do index seeks on bounded ranges works even with the 2D and 3D spatial `Point` types. Any following (non-existence check) predicates (here on property `p.firstname` for index `
:Person(place,firstname)`) will be rewritten as existence check with a filter. For index `
:Person(firstname,place)`, if the predicate on `firstname` is equality or list membership then the bounded range is handled as a range itself. If the predicate on `firstname` is anything else then the bounded range is rewritten to existence and filter.

Query

```cypher
MATCH (person:Person) WHERE point({x: 1, y: 5}) < person.place < point({x: 2, y: 6}) AND exists(person.firstname) RETURN person
```
Query Plan

Compiler CYPHER 4.0
Planner COST
Runtime PIPELINED
Runtime version 4.0

<table>
<thead>
<tr>
<th>Operator</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Time (ms)</th>
<th>Variables</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.151</td>
<td>person</td>
<td></td>
</tr>
<tr>
<td>+NodeIndexSeek(range,exists)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0.765</td>
<td>person</td>
<td></td>
</tr>
</tbody>
</table>

Total database accesses: 2

13.4. Basic query tuning example

We'll start with a basic example to help you get the hang of profiling queries. The following examples will use a movies data set.

Let’s start by importing the data:

```cypher
LOAD CSV WITH HEADERS FROM 'null/csv/query-tuning/movies.csv' AS line
MERGE (m:Movie { title: line.title })
ON CREATE SET m.released = toInteger(line.released), m.tagline = line.tagline
```

```cypher
LOAD CSV WITH HEADERS FROM 'null/csv/query-tuning/actors.csv' AS line
MATCH (m:Movie { title: line.title })
MERGE (p:Person { name: line.name })
ON CREATE SET p.born = toInteger(line.born)
MERGE (p)-[:ACTED_IN { roles:split(line.roles, ';')}]-(m)
```

```cypher
LOAD CSV WITH HEADERS FROM 'null/csv/query-tuning/directors.csv' AS line
MATCH (m:Movie { title: line.title })
MERGE (p:Person { name: line.name })
ON CREATE SET p.born = toInteger(line.born)
MERGE (p)-[:DIRECTED]-(m)
```

Let’s say we want to write a query to find 'Tom Hanks'. The naive way of doing this would be to write the following:

```cypher
MATCH (p { name: 'Tom Hanks' })
RETURN p
```

This query will find the 'Tom Hanks' node but as the number of nodes in the database increase it will become slower and slower. We can profile the query to find out why that is.
You can learn more about the options for profiling queries in Query tuning but in this case we’re going to prefix our query with PROFILE:

```cypher
PROFILE
MATCH (p { name: 'Tom Hanks' })
RETURN p
```

The first thing to keep in mind when reading execution plans is that you need to read from the bottom up.

In that vein, starting from the last row, the first thing we notice is that the value in the `Rows` column seems high given there is only one node with the name property 'Tom Hanks' in the database. If we look across to the `Operator` column we’ll see that `AllNodesScan` has been used which means that the query planner scanned through all the nodes in the database.

This seems like an inefficient way of finding 'Tom Hanks' given that we are looking at many nodes that aren’t even people and therefore aren’t what we’re looking for.

The solution to this problem is that whenever we’re looking for a node we should specify a label to help the query planner narrow down the search space. For this query we’d need to add a `Person` label.

```cypher
MATCH (p:Person { name: 'Tom Hanks' })
RETURN p
```

This query will be faster than the first one but as the number of people in our database increase we again notice that the query slows down.

Again we can profile the query to work out why:
This time the Rows value on the last row has reduced so we’re not scanning some nodes that we were before which is a good start. The NodeByLabelScan operator indicates that we achieved this by first doing a linear scan of all the Person nodes in the database.

Once we’ve done that we again scan through all those nodes using the Filter operator, comparing the name property of each one.

This might be acceptable in some cases but if we’re going to be looking up people by name frequently then we’ll see better performance if we create an index on the name property for the Person label:

```cypher
CREATE INDEX FOR (p:Person)
ON (p.name)
```

Now if we run the query again it will run more quickly:

```cypher
MATCH (p:Person { name: 'Tom Hanks' })
RETURN p
```

Let’s profile the query to see why that is:
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.

13.5. Advanced query tuning example

This section describes some more subtle optimizations based on new native index capabilities

- Introduction
- The data set
- Index-backed property-lookup
  - Aggregating functions
- Index-backed order by
  - \texttt{min()} and \texttt{max()}
  - Restrictions

13.5.1. Introduction

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. In Neo4j 3.4 and earlier, the fact
that the index contains the property value, and the results are returned in a specific order, was not used
improve the performance of any later part of the query that might depend on the property value or result
order.

Let's explain how to use these features with a more advanced query tuning example.

If you are upgrading an existing store to 4.0.12, 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.

13.5.2. The data set

In this example we will demonstrate the impact native indexes can have on query performance under
certain conditions. We'll use a movies dataset to illustrate this more advanced query tuning.

```cypher
LOAD CSV WITH HEADERS FROM 'null/csv/query-tuning/movies.csv' AS line
MERGE (m:Movie { title: line.title })
ON CREATE SET m.released = toInteger(line.released), m.tagline = line.tagline

LOAD CSV WITH HEADERS FROM 'null/csv/query-tuning/actors.csv' AS line
MATCH (m:Movie { title: line.title })
MERGE (p:Person { name: line.name })
ON CREATE SET p.born = toInteger(line.born)
MERGE (p)-[:ACTED_IN { roles:split(line.roles, ';')}]-(m)

LOAD CSV WITH HEADERS FROM 'null/csv/query-tuning/directors.csv' AS line
MATCH (m:Movie { title: line.title })
MERGE (p:Person { name: line.name })
ON CREATE SET p.born = toInteger(line.born)
MERGE (p)-[:DIRECTED]-(m)

CREATE INDEX FOR (p:Person)
ON (p.name)

CALL db.awaitIndexes

CALL db.indexes
```

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>state</th>
<th>populationPercent</th>
<th>uniqueness</th>
<th>type</th>
<th>entityType</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&quot;index_5c0607ad&quot;</td>
<td>&quot;ONLINE&quot;</td>
<td>100.0</td>
<td>&quot;NONUNIQUE&quot;</td>
<td>&quot;BTREE&quot;</td>
<td>&quot;NODE&quot;</td>
</tr>
<tr>
<td>[&quot;name&quot;]</td>
<td>&quot;native-btree-1.0&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
13.5.3. Index-backed property-lookup

Let’s say we 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, count(m)
```

<table>
<thead>
<tr>
<th>p.name</th>
<th>count(m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Tom Cruise&quot;</td>
<td>3</td>
</tr>
<tr>
<td>&quot;Tom Hanks&quot;</td>
<td>12</td>
</tr>
<tr>
<td>&quot;Tom Skerritt&quot;</td>
<td>1</td>
</tr>
</tbody>
</table>

3 rows

We have asked the database to return all the actors with the first name 'Tom'. There are three of them: 'Tom Cruise', 'Tom Skerritt' and 'Tom Hanks'. In previous versions of Neo4j, the final clause `RETURN p.name` would cause the database to take the node `p` and look up its properties and return the value of the property `name`. With native indexes, however, we 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 `NodeIndexScan` in the `Variables` column contains `cache[p.name]`, which means that `p.name` is retrieved from the index. We can also see that the `Projection` has no `DB Hits`, which means it does not have to access the database again.
### Compiler CYPHER 4.0
### Planner COST
### Runtime INTERPRETED
### Runtime version 4.0

| Operator              | Estimated Rows | Rows | DB Hits | Page Cache Hits | Page Cache Misses | Page Cache Hit Ratio | Order      | Variables        | Other               |
|-----------------------|----------------+------|--------|-----------|-----------------|-------------------|----------------------|------------|------------------|---------------------|
| +ProduceResults       |                1 |  3   |    0    |            0  |                 0 |                   0 | p.name ASC |                      |
|                       |                +----------------+------+---------+-----------------+-------------------|
| +OrderedAggregation   |                1 |  3   |    0    |            0  |                 0 |                   0 | p.name ASC |                      |
|                       |                +----------------+------+---------+-----------------+-------------------|
| +Filter               |                1 |  16  |     16  |            0  |                 0 |                   0 | p.name ASC |                      |
|                       |                +----------------+------+---------+-----------------+-------------------|
| +Expand(All)          |                1 |  16  |     20  |            0  |                 0 |                   0 | p.name ASC |                      |
|                       |                +----------------+------+---------+-----------------+-------------------|
| +NodeIndexSeekByRange |                1 |  4   |     5   |            0  |                 0 |                   0 | p.name ASC |                      |

Total database accesses: 41

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 Variables, and that the Projection now has DB Hits, since it accesses the database again to retrieve the name.

**PROFILE**

```cypher
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
RETURN p.name, count(m)
```
### Compiler

- **CYPHER 4.0**

### Planner

- **COST**

### Runtime

- **INTERPRETED**

**Runtime version 4.0**

| Operator          | Estimated Rows | Rows | DB Hits | Page Cache Hits | Page Cache Misses | Page Cache Hit Ratio | Variables | Other                |
|-------------------|----------------+------|--------|--------------|-----------------|-------------------|---------------------|-----------|---------------------|
| +ProduceResults   | 13 | 102 | 0 | 0 | 0 | 0.0000 | count(m), p.name |                      |
| +EagerAggregation | 13 | 102 | 172 | 0 | 0 | 0.0000 | count(m), p.name | p.name               |
| +Filter           | 172 | 172 | 172 | 0 | 0 | 0.0000 | anon[17], m, p | p:Person             |
| +Expand(All)      | 172 | 172 | 210 | 0 | 0 | 0.0000 | anon[17], p -- m | (m)<-[[:ACTED_IN]]-(p) |
| +NodeByLabelScan  | 38 | 38 | 39 | 0 | 0 | 0.0000 | m | :Movie               |

**Total database accesses: 593**

It is important to note that not all property types are supported, because not all are supported by native indexes. Additionally, some property types such as the spatial type `Point`, are indexed in an index that is designed to be approximate and cannot return the values. For non-native indexes and the spatial type `Point`, there will still be a second database access to retrieve those values. In indexes with mixed values, only those values that cannot be looked up from the index will trigger another database access action.

**Predicates that can be used to enable this optimization are:**

- **Existence** (WHERE `exists(n.name)`)
- **Equality** (e.g. WHERE `n.name = 'Tom Hanks'`)
- **Range** (eg. WHERE `n.uid > 1000 AND n.uid < 2000`)
- **Prefix** (eg. WHERE `n.name STARTS WITH 'Tom'`)
- **Suffix** (eg. WHERE `n.name ENDS WITH 'Hanks'`)
- **Substring** (eg. WHERE `n.name CONTAINS 'a'`)
- **Several predicates of the above types combined using OR**, given that all of them are on the same property (eg. 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 ON (p:Person) ASSERT exists(p.name)`
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
```

Compiler CYPHER 4.0
Planner COST
Runtime INTERPRETED
Runtime version 4.0

| Operator          | Estimated Rows | Rows | DB Hits | Page Cache Hits | Page Cache Misses | Page Cache Hit Ratio | Variables     | Other                        |
|-------------------+----------------+------+---------+-----------------+-------------------+---------------------+---------------+------------------------------|
| +ProduceResults   |              1 |    1 |       0 |               0 |                 0 | 0.0000 | numberOfNames |                              |
| +EagerAggregation |              1 |    1 |       0 |               0 |                 0 | 0.0000 | numberOfNames |                              |
| +NodeIndexScan    |            125 |  125 |     126 |               0 |                 0 | 0.0000 | p             | :Person(name), cache[p.name] |

Total database accesses: 126

Note that the NodeIndexScan in the Variables 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 constraint because Person nodes without the property name will not affect the result of an aggregation.

13.5.4. Index-backed order by

Now consider the following refinement to the query:

```
MATCH (p:Person)-[r:ACTED_IN]-(m:Movie)
WHERE p.name STARTS WITH 'Tom'
RETURN p.name, count(m)
ORDER BY p.name
```

<table>
<thead>
<tr>
<th>p.name</th>
<th>count(m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Tom Cruise&quot;</td>
<td>3</td>
</tr>
<tr>
<td>&quot;Tom Hanks&quot;</td>
<td>12</td>
</tr>
<tr>
<td>&quot;Tom Skerritt&quot;</td>
<td>1</td>
</tr>
</tbody>
</table>

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 3.4 and earlier, Cypher would plan a **Sort** operation to sort the results, which means building a collection in memory and running a sort algorithm on it. For large result sets this can be expensive in terms of both memory and time. If you are using the native index, Cypher will recognise that the index already returns data in the correct order, and skip the **Sort** operation.

Indexes storing values of the spatial type **Point**, and non-native indexes, cannot be relied on to return the values in the correct order. This means that for Cypher to enable this optimization, the query needs a predicate that limits the type of the property to some type that is guaranteed to be in the right order.

To demonstrate this effect, let’s remove the String prefix predicate so that Cypher no longer knows the type of the property, and replace it with an existence predicate. Now the database can no longer guarantee the order. If we profile the query we will see the **Sort** operation:

```cypher
PROFILE
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
USING INDEX p:Person(name)
WHERE EXISTS (p.name)
RETURN p.name, count(m)
ORDER BY p.name
```

<table>
<thead>
<tr>
<th>Operator</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Page Cache Hits</th>
<th>Page Cache Misses</th>
<th>Page Cache Hit Ratio</th>
<th>Order</th>
<th>Variables</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>13</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>p.name ASC</td>
<td>count(m), p.name</td>
<td></td>
</tr>
<tr>
<td>+Sort</td>
<td>13</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>p.name ASC</td>
<td>count(m), p.name</td>
<td>p.name</td>
</tr>
<tr>
<td>+EagerAggregation</td>
<td>13</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>count(m), p.name</td>
<td>p.name</td>
<td>p.name</td>
</tr>
<tr>
<td>+Filter</td>
<td>172</td>
<td>172</td>
<td>172</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>anon[17], m, p</td>
<td>m:Movie</td>
<td></td>
</tr>
<tr>
<td>+Expand(All)</td>
<td>172</td>
<td>172</td>
<td>297</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>anon[17], m -- p</td>
<td>(p)-[:ACTED_IN]-&gt;(m)</td>
<td></td>
</tr>
<tr>
<td>+NodeIndexScan</td>
<td>125</td>
<td>125</td>
<td>126</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>p</td>
<td>:Person(name), cache[p.name]</td>
<td></td>
</tr>
</tbody>
</table>

Total database accesses: 595

The **Order** column describes the order of rows after each operator. We see that the order is undefined until
the Sort operator. Now if we add back the predicate that gives us the property type information, we will see the Sort operation is no longer there:

```cypher
PROFILE
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WHERE p.name STARTS WITH 'Tom'
RETURN p.name, count(m)
ORDER BY p.name
```

We also see that the Order column contains `p.name ASC` from the index seek operation, meaning that the rows are ordered by `p.name` in ascending order.

Index-backed order by can also be used for queries that expect their results is descending order, but with slightly lower performance.

In cases where the Cypher planner is unable to remove the Sort operator, like in the first example, the planner can utilize knowledge of required order after each operator to plan the Sort operator at a point in the plan with optimal cardinality.
For the \texttt{min} and \texttt{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:

\begin{verbatim}
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WHERE p.name STARTS WITH ''
RETURN min(p.name) AS name
\end{verbatim}

\begin{verbatim}
+----------------+
| name           |
+----------------+
| "Aaron Sorkin" |
+----------------+
1 row
\end{verbatim}

To demonstrate the effect of index-backed order by, let’s remove the String prefix predicate so that Cypher no longer knows the type of the property, and replace it with an existence predicate. Now the database can no longer guarantee the order. If we profile the query we will see the \texttt{EagerAggregation} operation:

\begin{verbatim}
PROFILE
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
USING INDEX p:Person(name)
WHERE EXISTS (p.name)
RETURN min(p.name) AS name
\end{verbatim}

\begin{verbatim}
Compiler CYPHER 4.0
Planner COST
Runtime INTERPRETED
Runtime version 4.0

| Operator          | Estimated Rows | Rows | DB Hits | Page Cache Hits | Page Cache Misses | Page Cache Hit Ratio | Variables | Other |
|-------------------|----------------+------+---------+-----------------+-------------------+---------------------+-----------+-------|
| +ProduceResults   | 1              | 1    | 0       | 0               | 0                 | 0.0000              | name      |       |
| +EagerAggregation | 1              | 1    | 0       | 0               | 0                 | 0.0000              | name      |       |
| +Filter           | 172            | 172  | 172     | 0               | 0                 | 0.0000              | anon[17], m, p | m:Movie |
| +Expand(All)      | 172            | 172  | 297     | 0               | 0                 | 0.0000              | anon[17], m | (p)-[:ACTED_IN]->(m) |
| +NodeIndexScan    | 125            | 125  | 126     | 0               | 0                 | 0.0000              | p | :Person(name), cache[p.name] |

Total database accesses: 595
\end{verbatim}
Now if we add back the predicate that gives us the property type information, we will see that the EagerAggregation operation gets replaced by Projection followed by Limit followed by Optional:

```
PROFILE
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WHERE p.name STARTS WITH ''
RETURN min(p.name) AS name
```

```
| Operator              | Estimated Rows | Rows | DB Hits | Page Cache Hits | Page Cache Misses | Page Cache Hit Ratio | Order      | Variables              | Other |
|-----------------------+----------------+------|--------+----------------+-------------------+---------------------+------------+------------------------+-------|
| +ProduceResults       |              1 |    1 |       0 |               0 |                 0 |                   | 0.0000 | name ASC   | anon[17], m, name, p   |       |
| |                     +----------------+------+---------+-----------------+-------------------+-----------------------------------------------------------+| |
| +Optional             |              1 |    1 |       0 |               0 |                 0 |                   | 0.0000 | name ASC   | anon[17], m, name, p   |       |
| |                     +----------------+------+---------+-----------------+-------------------+-----------------------------------------------------------+| |
| +Limit                |              1 |    1 |       0 |               0 |                 0 |                   | 0.0000 | name ASC   | anon[17], m, name, p   |       |
| |                     +----------------+------+---------+-----------------+-------------------+-----------------------------------------------------------+| |
| +Projection           |              1 |    1 |       0 |               0 |                 0 |                   | 0.0000 | p.name ASC | anon[17], m, p         |       |
| |                     +----------------+------+---------+-----------------+-------------------+-----------------------------------------------------------+| |
| +Filter               |              1 |    1 |       0 |               0 |                 0 |                   | 0.0000 | p.name ASC | anon[17], m         |       |
| |                     +----------------+------+---------+-----------------+-------------------+-----------------------------------------------------------+| |
| +Expand(All)          |              1 |    1 |       0 |               0 |                 0 |                   | 0.0000 | p.name ASC | anon[17], m, p       |       |
| |                     +----------------+------+---------+-----------------+-------------------+-----------------------------------------------------------+| |
| +NodeIndexSeekByRange |              1 |    1 |       0 |               0 |                 0 |                   | 0.0000 | p.name ASC | p                      |       |
| |                     +----------------+------+---------+-----------------+-------------------+-----------------------------------------------------------+| |
```

In the first case, all nodes in the index are scanned to find the name that is first in alphabetic order. In the second case, we will simply pick the first value from the index. This is reflected in the fact that the total database access is lower, indicating a faster query. 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
Restrictions

The optimization can only work on native indexes, and only if we query for a specific type, in order to rule out the spatial type `Point`. Predicates that can be used to enable this optimization are:

- 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` because the property type might differ between the predicates
- Existence (e.g. `WHERE exists(n.email)`) because no property type information is given

If the predicate uses a parameter, such as `WHERE n.prop > $param`, it will not trigger index-backed order by in this version of Neo4j.

An existence constraint does not include any type information either and will thus not be enough to trigger index-backed order by.

13.6. Planner hints and the USING keyword

A planner hint is used to influence the decisions of the planner when building an execution plan for a query. Planner hints are specified in a query with the `USING` keyword.

Forcing planner behavior is an advanced feature, and should be used with caution by experienced developers and/or database administrators only, as it may cause queries to perform poorly.

13.6.1. Introduction

When executing a query, Neo4j needs to decide where in the query graph to start matching. This is done by looking at the `MATCH` clause and the `WHERE` conditions and using that information to find useful indexes,
or other starting points.

However, the selected index might not always be the best choice. Sometimes multiple indexes are possible candidates, and the query planner picks the suboptimal one from a performance point of view. Moreover, in some circumstances (albeit rarely) it is better not to use an index at all.

Neo4j can be forced to use a specific starting point through the `USING` keyword. This is called giving a planner hint. There are four types of planner hints: index hints, scan hints, join hints, and the `PERIODIC COMMIT` query hint.

The following graph is used for the examples below:

![Graph]

**Graph**

**Query**

```cypher
MATCH (liskov:Scientist {name: 'Liskov'})-[[:KNOWS]]->(wing:Scientist)-[[:RESEARCHED]]->(cs:Science { name: 'Computer Science'})<-[[:RESEARCHED]]-(conway:Scientist {name: 'Conway'}) RETURN 1 AS column
```

The following query will be used in some of the examples on this page. It has intentionally been constructed in such a way that the statistical information will be inaccurate for the particular path that this query matches. For this reason, it can be improved by supplying planner hints.

**Query plan**

```
Compiler CYPHER 4.0
Planner COST
Runtime PIPELINED
Runtime version 4.0
```

```
<table>
<thead>
<tr>
<th>Operator</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Time (ms)</th>
<th>Order</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Other</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ProduceResults</td>
<td></td>
<td>0</td>
<td>1</td>
<td>0.052</td>
<td>cs.name ASC</td>
<td>anon[122], anon[41], anon[68], column, conway, cs, liskov, wing</td>
</tr>
<tr>
<td>+Projection</td>
<td></td>
<td>0</td>
<td>1</td>
<td>0.014</td>
<td>cs.name ASC</td>
<td>column -- anon[122], anon[41], anon[68], conway, cs, liskov, wing</td>
</tr>
<tr>
<td>+NodeHashJoin</td>
<td></td>
<td>0</td>
<td>1</td>
<td>0.047</td>
<td>cs.name ASC</td>
<td>anon[122], anon[41], liskov</td>
</tr>
<tr>
<td>+Filter</td>
<td></td>
<td>0</td>
<td>2</td>
<td>0.102</td>
<td>anon[122], anon[68], conway, cs, wing</td>
<td></td>
</tr>
<tr>
<td>+NodeHashJoin</td>
<td></td>
<td>0</td>
<td>3</td>
<td>0.811</td>
<td>cs.name ASC</td>
<td>anon[68], wing</td>
</tr>
</tbody>
</table>
```
13.6.2. Index hints

Index hints are used to specify which index, if any, the planner should use as a starting point. This can be beneficial in cases where the index statistics are not accurate for the specific values that the query at hand is known to use, which would result in the planner picking a non-optimal index. To supply an index hint, use `USING INDEX variable:Label(property)` or `USING INDEX SEEK variable:Label(property)` after the applicable `MATCH` clause.

It is possible to supply several index hints, but keep in mind that several starting points will require the use of a potentially expensive join later in the query plan.

Query using an index hint

The query above will not naturally pick an index to solve the plan. This is because the graph is very small, and label scans are faster for small databases. In general, however, query performance is ranked by the `dbhit` metric, and we see that using an index is slightly better for this query.
Query plan

```cyp
MATCH (liskov:Scientist {name: 'Liskov'})-[[:KNOWS]->(wing:Scientist)]->(cs:Science {name: 'Computer Science'})-[[:RESEARCHED]-(conway:Scientist {name: 'Conway'})]
USING INDEX liskov:Scientist (name)
RETURN liskov.born AS column
```

Returns the year 'Barbara Liskov' was born.
Query using an index seek hint

Similar to the index (scan) hint, but an index seek will be used rather than an index scan. Index seeks require no post filtering, they are most efficient when a relatively small number of nodes have the specified value on the queried property.

Query

```
MATCH (liskov:Scientist {name: 'Liskov'})-[[:KNOWS]->(wing):Scientist]-[:RESEARCHED]->(cs:Science {name: 'Computer Science'})<-[[:RESEARCHED]-(conway:Scientist {name: 'Conway'})] USING INDEX SEEK
liskov:Scientist(name) RETURN liskov.born AS column
```

Returns the year 'Barbara Liskov' was born.

Query plan

```
Compiler CYPHER 4.0
Planner COST
Runtime PIPELINED
Runtime version 4.0
```

```
<table>
<thead>
<tr>
<th>Operator</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Time (ms)</th>
<th>Order</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expand(All)</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
<td>liskov.name ASC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(liskov)-[:KNOWS]-&gt;(wing)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NodeIndexSeek</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td>liskov.name ASC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NodeIndexSeek</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td>liskov.name ASC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| Total database accesses: 28
```
Query using multiple index hints

Supplying one index hint changed the starting point of the query, but the plan is still linear, meaning it only has one starting point. If we give the planner yet another index hint, we force it to use two starting points, one at each end of the match. It will then join these two branches using a join operator.

Query

```
MATCH
  (liskov:Scientist {name: 'Liskov'})-[[:KNOWS]]-(wing:Scientist)
  <-[[:RESEARCHED]]-(cs:Science {name: 'Computer Science'})<-[[:RESEARCHED]]-
  (conway:Scientist {name: 'Conway'})
USING INDEX liskov:Scientist (name) USING INDEX conway:Scientist (name)
RETURN liskov.born AS column
```

Returns the year 'Barbara Liskov' was born, using a slightly better plan.

Query plan

```
Compiler CYPHER 4.0
Planner COST
Runtime PIPELINED
Runtime version 4.0
<table>
<thead>
<tr>
<th>Operator</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Time (ms)</th>
<th>Order</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Other</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ProduceResults</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.065</td>
<td></td>
<td>anon[122], anon[41], anon[68], column, conway, cs, liskov, wing</td>
</tr>
<tr>
<td>+NodeIndexSeek</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0.125</td>
<td></td>
<td>conway.name ASC, conway:Scientist (name)</td>
</tr>
<tr>
<td>+NodeIndexSeek</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0.097</td>
<td></td>
<td>cs:Science (name)</td>
</tr>
<tr>
<td>+Filter</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td>cs.name ASC, cs:Science (name)</td>
</tr>
<tr>
<td>+Expand(All)</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td>(cs)&lt;-[[:RESEARCHED]]-(wing), cs:Science (name)</td>
</tr>
<tr>
<td>+Filter</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>liskov.name ASC, anon[41], liskov:Scientist (name)</td>
</tr>
<tr>
<td>+Expand(All)</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
<td>(liskov)-[:KNOWS]-&gt;(wing), liskov:Scientist (name)</td>
</tr>
<tr>
<td>+NodeIndexSeek</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td>liskov.name ASC, liskov:Scientist (name)</td>
</tr>
</tbody>
</table>
```

Total database accesses: 28
If the best performance is to be had by scanning all nodes in a label and then filtering on that set, use hinting a label scan will force Cypher to not use an index that could have been used, and instead do a label scan.

To do this, you can use

```cypher
USING SCAN variable:Label
```

after the applicable clause. This will force Cypher to not use an index that could have been used, and instead do a label scan.

13.6.3. Scan hints

If your query matches large parts of an index, it might be faster to scan the label and filter out nodes that do not match. To do this, you can use `USING SCAN variable:Label` after the applicable `MATCH` clause. This will force Cypher to not use an index that could have been used, and instead do a label scan.

Hinting a label scan

If the best performance is to be had by scanning all nodes in a label and then filtering on that set, use
USING SCAN.

Query

```
MATCH (s:Scientist)
USING SCAN s:Scientist
WHERE s.born < 1939
RETURN s.born AS column
```

Returns all scientists born before '1939'.

Query plan

```
Compiler CYPHER 4.0
Planner COST
Runtime PIPELINED
Runtime version 4.0
```

<table>
<thead>
<tr>
<th>Operator</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Variables</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td></td>
<td>3</td>
<td>2</td>
<td>column, s</td>
<td></td>
</tr>
<tr>
<td>+Projection</td>
<td></td>
<td>3</td>
<td>2</td>
<td>column -- s</td>
<td>{column : cache[s.born]}</td>
</tr>
<tr>
<td>+Filter</td>
<td></td>
<td>3</td>
<td>2</td>
<td>18</td>
<td>s</td>
</tr>
<tr>
<td>+NodeByLabelScan</td>
<td>10</td>
<td>7</td>
<td>8</td>
<td>s</td>
<td>:Scientist</td>
</tr>
</tbody>
</table>

Total database accesses: 26

13.6.4. Join hints

Join hints are the most advanced type of hints, and are not used to find starting points for the query execution plan, but to enforce that joins are made at specified points. This implies that there has to be more than one starting point (leaf) in the plan, in order for the query to be able to join the two branches ascending from these leaves. Due to this nature, joins, and subsequently join hints, will force the planner to look for additional starting points, and in the case where there are no more good ones, potentially pick a very bad starting point. This will negatively affect query performance. In other cases, the hint might force the planner to pick a seemingly bad starting point, which in reality proves to be a very good one.

Hinting a join on a single node

In the example above using multiple index hints, we saw that the planner chose to do a join on the cs node. This means that the relationship between wing and cs was traversed in the outgoing direction, which is better statistically because the pattern ( ):RESEARCHED( :Science) is more common than the pattern ( :Scientist)-:RESEARCHED( ). However, in the actual graph, the cs node only has two such relationships, so expanding from it will be beneficial to expanding from the wing node. We can force the join to happen on wing instead with a join hint.
Returns the birth date of 'Jeanette Wing', using a slightly better plan.

**Query**

```cypher
MATCH (liskov:Scientist {name: 'Liskov'})-[[:KNOWS]]->(wing:Scientist)-[[:RESEARCHED]]->(cs:Science {name: 'Computer Science'}) AS cs
JOIN INDEX liskov:Scientist
USING INDEX conway:Scientist
USING JOIN ON wing
RETURN wing.born AS column
```

**Query plan**

```
MATCH (liskov:Scientist {name: 'Liskov'})-[[:KNOWS]]->(wing:Scientist)-[[:RESEARCHED]]->(cs:Science {name: 'Computer Science'}) AS cs
JOIN INDEX liskov:Scientist
USING INDEX conway:Scientist
USING JOIN ON wing
RETURN wing.born AS column
```
Hinting a join on multiple nodes

The query planner can be made to produce a join between several specific points. This requires the query to expand from the same node from several directions.

Query

MATCH (liskov:Scientist {name: 'Liskov'})-[[:KNOWS]->(wing:Scientist {name: 'Wing'})]-[:RESEARCHED]->(cs:Science {name: 'Computer Science'})<-[:RESEARCHED]-(liskov)
USING INDEX liskov:Scientist(name)
USING JOIN ON liskov, cs
RETURN wing.born AS column

Returns the birth date of 'Jeanette Wing'.

Query plan
| \ | +NodeIndexSeek | 0 | 1 | 2 | 1 | 0 |
| \ | +Expand(All) | 0 | 2 | 3 | 1 | 0 |
| +NodeHashJoin | | liskov.name ASC | anon[41] -- liskov, wing | (liskov)-[:KNOWS]->(cs) |
| +Expand(All) | | wing.name ASC | anon[136] -- cs, wing | (cs)<-[[:RESEARCHED]]-(cs) |
| | +NodeIndexSeek | 0 | 1 | 2 | 1 | 0 |
| | +NodeIndexSeek | 0 | 1 | 2 | 1 | 0 |
| | +NodeIndexSeek | 0 | 1 | 2 | 1 | 0 |
| | +Expand(All) | 0 | 1 | 3 | 1 | 0 |
| +Filter | cache[cs.name] = $` AUTOSTRING2'; cs:Science |
| +NodeHashJoin | al | wing.name ASC | anon[41], liskov -- anon[82], cs, wing | (wing)-[:RESEARCHED]->(cs) |
| +Expand(All) | | wing.name ASC | anon[136] -- cs, liskov | (cs)<-[[:RESEARCHED]]-(liskov) |
| +CartesianProduct | 0 | 1 | 0 | 0 | 0 |
| +NodeHashJoin | wing.name ASC | anon[41], liskov -- anon[82], cs, wing | (liskov)-[:KNOWS]->(cs) |
| +Expand(All) | | liskov.name ASC | anon[82], cs -- wing | (liskov)-[:KNOWS]->(liskov) |
| +CartesianProduct | | liskov.name ASC | anon[41], liskov -- anon[82], cs, wing | (liskov)-[:KNOWS]->(cs) |
| +Expand(All) | | liskov.name ASC | anon[82], cs -- wing | (liskov)-[:KNOWS]->(cs) |
| +NodeIndexSeek | | 0 | 1 | 2 | 1 | 0 |
| +Expand(All) | | wing.name ASC | wing | (wing)-[:RESEARCHED]->(cs) |
| +NodeIndexSeek | | wing | anonymous[128] -- cs, liskov | (cs)<-[[:RESEARCHED]]-(cs) |
| +Expand(All) | | liskov.name ASC | liskov -- anon[82], cs, wing | (liskov)-[:KNOWS]->(cs) |
| +NodeHashJoin | wing.name ASC | anonymous[128], liskov -- anonymous[82], cs, wing | (liskov)-[:KNOWS]->(cs) |
| +Expand(All) | | liskov.name ASC | anonymous[82], cs -- wing | (liskov)-[:KNOWS]->(cs) |
| +NodeIndexSeek | | 0 | 1 | 2 | 1 | 0 |
| +Expand(All) | | wing.name ASC | wing | (wing)-[:RESEARCHED]->(cs) |
| +NodeIndexSeek | | wing | anonymous[128] -- cs, liskov | (cs)<-[[:RESEARCHED]]-(cs) |
| +Expand(All) | | liskov.name ASC | liskov -- anon[82], cs, wing | (liskov)-[:KNOWS]->(cs) |
| +NodeHashJoin | wing.name ASC | anonymous[128], liskov -- anonymous[82], cs, wing | (liskov)-[:KNOWS]->(cs) |
| +Expand(All) | | liskov.name ASC | liskov -- anon[82], cs, wing | (liskov)-[:KNOWS]->(cs) |
| +NodeIndexSeek | | 0 | 1 | 2 | 1 | 0 |
| +Expand(All) | | wing.name ASC | wing | (wing)-[:RESEARCHED]->(cs) |
| +NodeIndexSeek | | wing | anonymous[128] -- cs, liskov | (cs)<-[[:RESEARCHED]]-(cs) |
| +Expand(All) | | liskov.name ASC | liskov -- anon[82], cs, wing | (liskov)-[:KNOWS]->(cs) |
| +NodeHashJoin | wing.name ASC | anonymous[128], liskov -- anonymous[82], cs, wing | (liskov)-[:KNOWS]->(cs) |
| +Expand(All) | | liskov.name ASC | liskov -- anon[82], cs, wing | (liskov)-[:KNOWS]->(cs) |
| +NodeIndexSeek | | 0 | 1 | 2 | 1 | 0 |
| +Expand(All) | | wing.name ASC | wing | (wing)-[:RESEARCHED]->(cs) |
| +NodeIndexSeek | | wing | anonymous[128] -- cs, liskov | (cs)<-[[:RESEARCHED]]-(cs) |
| +Expand(All) | | liskov.name ASC | liskov -- anon[82], cs, wing | (liskov)-[:KNOWS]->(cs) |
| +NodeHashJoin | wing.name ASC | anonymous[128], liskov -- anonymous[82], cs, wing | (liskov)-[:KNOWS]->(cs) |
| +Expand(All) | | liskov.name ASC | liskov -- anon[82], cs, wing | (liskov)-[:KNOWS]->(cs) |
| +NodeIndexSeek | | 0 | 1 | 2 | 1 | 0 |
| +Expand(All) | | wing.name ASC | wing | (wing)-[:RESEARCHED]->(cs) |
| +NodeIndexSeek | | wing | anonymous[128] -- cs, liskov | (cs)<-[[:RESEARCHED]]-(cs) |
| +Expand(All) | | liskov.name ASC | liskov -- anon[82], cs, wing | (liskov)-[:KNOWS]->(cs) |
| +NodeHashJoin | wing.name ASC | anonymous[128], liskov -- anonymous[82], cs, wing | (liskov)-[:KNOWS]->(cs) |
| +Expand(All) | | liskov.name ASC | liskov -- anon[82], cs, wing | (liskov)-[:KNOWS]->(cs) |
| +NodeIndexSeek | | 0 | 1 | 2 | 1 | 0 |
| +Expand(All) | | wing.name ASC | wing | (wing)-[:RESEARCHED]->(cs) |
| +NodeIndexSeek | | wing | anonymous[128] -- cs, liskov | (cs)<-[[:RESEARCHED]]-(cs) |
| +Expand(All) | | liskov.name ASC | liskov -- anon[82], cs, wing | (liskov)-[:KNOWS]->(cs) |
| +NodeHashJoin | wing.name ASC | anonymous[128], liskov -- anonymous[82], cs, wing | (liskov)-[:KNOWS]->(cs) |
| +Expand(All) | | liskov.name ASC | liskov -- anon[82], cs, wing | (liskov)-[:KNOWS]->(cs) |
13.6.5. **PERIODIC COMMIT** query hint

Importing large amounts of data using `LOAD CSV` with a single Cypher query may fail due to memory constraints. This will manifest itself as an `OutOfMemoryError`.

For this situation only, Cypher provides the global `USING PERIODIC COMMIT` query hint for updating queries using `LOAD CSV`. If required, the limit for the number of rows per commit may be set as follows: `USING PERIODIC COMMIT 500`.

`PERIODIC COMMIT` will process the rows until the number of rows reaches a limit. Then the current transaction will be committed and replaced with a newly opened transaction. If no limit is set, a default value will be used.

See [Importing large amounts of data](#) in `LOAD CSV` for examples of **USING PERIODIC COMMIT** with and without setting the number of rows per commit.

| ![Warning] | Using **PERIODIC COMMIT** will prevent running out of memory when importing large amounts of data. However, it will also break transactional isolation and thus it should only be used where needed. |

---

458
Chapter 14. Execution plans

This section describes the characteristics of query execution plans and provides details about each of the operators.

- Introduction
- Execution plan operators
- Database hits (DbHits)
- Shortest path planning

14.1. Introduction

The task of executing a query is decomposed into operators, each of which implements a specific piece of work. The operators are combined into a tree-like structure called an execution plan. Each operator in the execution plan is represented as a node in the tree. Each operator takes as input zero or more rows, and produces as output zero or more rows. This means that the output from one operator becomes the input for the next operator. Operators that join two branches in the tree combine input from two incoming streams and produce a single output.

Evaluation model

Evaluation of the execution plan begins at the leaf nodes of the tree. Leaf nodes have no input rows and generally comprise operators such as scans and seeks. These operators obtain the data directly from the storage engine, thus incurring database hits. Any rows produced by leaf nodes are then piped into their parent nodes, which in turn pipe their output rows to their parent nodes and so on, all the way up to the root node. The root node produces the final results of the query.

Eager and lazy evaluation

In general, query evaluation is lazy: most operators pipe their output rows to their parent operators as soon as they are produced. This means that a child operator may not be fully exhausted before the parent operator starts consuming the input rows produced by the child.

However, some operators, such as those used for aggregation and sorting, need to aggregate all their rows before they can produce output. Such operators need to complete execution in its entirety before any rows are sent to their parents as input. These operators are called eager operators, and are denoted as such in Execution plan operators at a glance. Eagerness can cause high memory usage and may therefore be the cause of query performance issues.

Statistics

Each operator is annotated with statistics.

Rows

The number of rows that the operator produced. This is only available if the query was profiled.
**EstimatedRows**

This is the estimated number of rows that is expected to be produced by the operator. The estimate is an approximate number based on the available statistical information. The compiler uses this estimate to choose a suitable execution plan.

**DbHits**

Each operator will ask the Neo4j storage engine to do work such as retrieving or updating data. A database hit is an abstract unit of this storage engine work. The actions triggering a database hit are listed in Database hits (DbHits).

**Page Cache Hits, Page Cache Misses, Page Cache Hit Ratio**

These metrics are only shown for some queries when using Neo4j Enterprise Edition. The page cache is used to cache data and avoid accessing disk, so having a high number of hits and a low number of misses will typically make the query run faster.

**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.

To produce an efficient plan for a query, the Cypher query planner requires information about the Neo4j database. This information includes which indexes and constraints are available, as well as various statistics maintained by the database. The Cypher query planner uses this information to determine which access patterns will produce the best execution plan.

The statistical information maintained by Neo4j is:

1. The number of nodes having a certain label.
2. The number of relationships by type.
3. Selectivity per index.
4. The number of relationships by type, ending with or starting from a node with a specific label.

Information about how the statistics are kept up to date, as well as configuration options for managing query replanning and caching, can be found in the Operations Manual → Statistics and execution plans.

Query tuning describes how to tune Cypher queries. In particular, see Query tuning for how to view the execution plan for a query and Planner hints and the USING keyword for how to use hints to influence the decisions of the planner when building an execution plan for a query.

For a deeper understanding of how each operator works, refer to Execution plan operators at a glance and the linked sections per operator. Please remember that the statistics of the particular database where the queries run will decide the plan used. There is no guarantee that a specific query will always be solved with the same plan.

**14.2. Execution plan operators at a glance**

This table comprises all the execution plan operators ordered lexicographically.

- Leaf operators, in most cases, locate the starting nodes and relationships required in order to execute
the query.

- **Updating operators** are used in queries that update the graph.
- **Eager operators** accumulate all their rows before piping them to the next operator.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Leaf?</th>
<th>Updating?</th>
<th>Considerations</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllNodesScan</td>
<td>Reads all nodes from the node store.</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AntiConditionalApply</td>
<td>Performs a nested loop. If a variable is null, the right-hand side will be executed.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AntiSemiApply</td>
<td>Performs a nested loop. Tests for the absence of a pattern predicate.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apply</td>
<td>Performs a nested loop. Yields rows from both the left-hand and right-hand side operators.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Argument</td>
<td>Indicates the variable to be used as an argument to the right-hand side of an Apply operator.</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AssertSameNode</td>
<td>Used to ensure that no unique constraints are violated.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CacheProperties</td>
<td>Reads node or relationship properties and caches them.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CartesianProduct</td>
<td>Produces a cartesian product of the inputs from the left-hand and right-hand operators.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConditionalApply</td>
<td>Performs a nested loop. If a variable is not null, the right-hand side will be executed.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CreateIndex</td>
<td>Creates an index on a property for all nodes having a certain label.</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CreateNodeKeyConstra int</td>
<td>Creates a node key constraint on a set of properties for all nodes having a certain label.</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Create</td>
<td>Creates nodes and relationships.</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Leaf?</td>
<td>Updating?</td>
<td>Considerations</td>
</tr>
<tr>
<td>----------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------</td>
<td>-----------</td>
<td>-----------------</td>
</tr>
<tr>
<td>CreateNodePropertyExistenceConstraint</td>
<td>Creates an existence constraint on a property for all nodes having a certain label.</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>CreateRelationshipPropertyExistenceConstraint</td>
<td>Creates an existence constraint on a property for all relationships of a certain type.</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>CreateUniqueConstraint</td>
<td>Creates a unique constraint on a property for all nodes having a certain label.</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Delete</td>
<td>Deletes a node or relationship.</td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>DetachDelete</td>
<td>Deletes a node and its relationships.</td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>DirectedRelationshipByIdSeek</td>
<td>Reads one or more relationships by id from the relationship store.</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distinct</td>
<td>Drops duplicate rows from the incoming stream of rows.</td>
<td></td>
<td></td>
<td>Eager</td>
</tr>
<tr>
<td>DropIndex (deprecated syntax)</td>
<td>Drops an index from a property for all nodes having a certain label.</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>DropIndex (new syntax)</td>
<td>Drops an index using its name.</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>DropConstraint</td>
<td>Drops a constraint using its name.</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>DropNodeKeyConstraint</td>
<td>Drops a node key constraint from a set of properties for all nodes having a certain label.</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>DropIndex</td>
<td>Drops a node property existence constraint from a property for all nodes having a certain label.</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>DropIndex</td>
<td>Drops a relationship property existence constraint from a property for all relationships of a certain type.</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Leaf?</td>
<td>Updating?</td>
<td>Considerations</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------</td>
<td>-----------</td>
<td>----------------</td>
</tr>
<tr>
<td>DropResult</td>
<td>Produces zero rows when an expression is guaranteed to produce an empty result.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DropUniqueConstraint</td>
<td>Drops a unique constraint from a property for all nodes having a certain label.</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Eager</td>
<td>For isolation purposes, <em>Eager</em> ensures that operations affecting subsequent operations are executed fully for the whole dataset before continuing execution.</td>
<td></td>
<td></td>
<td>Eager</td>
</tr>
<tr>
<td>EagerAggregation</td>
<td>Evaluates a grouping expression.</td>
<td></td>
<td></td>
<td>Eager</td>
</tr>
<tr>
<td>EmptyResult</td>
<td>Eagerly loads all incoming data and discards it.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EmptyRow</td>
<td>Returns a single row with no columns.</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expand(All)</td>
<td>Traverses incoming or outgoing relationships from a given node.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expand(Into)</td>
<td>Finds all relationships between two nodes.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Filter</td>
<td>Filters each row coming from the child operator, only passing through rows that evaluate the predicates to <em>true</em>.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Foreach</td>
<td>Performs a nested loop. Yields rows from the left-hand operator and discards rows from the right-hand operator.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LetAntiSemiApply</td>
<td>Performs a nested loop. Tests for the absence of a pattern predicate in queries containing multiple pattern predicates.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Leaf?</td>
<td>Updating?</td>
<td>Considerations</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------</td>
<td>-----------</td>
<td>----------------</td>
</tr>
<tr>
<td>LetSelectOrSemiApply</td>
<td>Performs a nested loop. Tests for the presence of a pattern predicate that is combined with other predicates.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LetSelectOrAntiSemiApply</td>
<td>Performs a nested loop. Tests for the absence of a pattern predicate that is combined with other predicates.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LetSemiApply</td>
<td>Performs a nested loop. Tests for the presence of a pattern predicate in queries containing multiple pattern predicates.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Limit</td>
<td>Returns the first 'n' rows from the incoming input.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LoadCSV</td>
<td>Loads data from a CSV source into the query.</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LockNodes</td>
<td>Locks the start and end node when creating a relationship.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MergeCreateNode</td>
<td>Creates the node when failing to find the node.</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>MergeCreateRelationship</td>
<td>Creates the relationship when failing to find the relationship.</td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>NodeByIdSeek</td>
<td>Reads one or more nodes by id from the node store.</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NodeByLabelScan</td>
<td>Fetches all nodes with a specific label from the node label index.</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NodeCountFromCountStore</td>
<td>Uses the count store to answer questions about node counts.</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NodeHashJoin</td>
<td>Executes a hash join on node ids.</td>
<td></td>
<td>Eager</td>
<td></td>
</tr>
<tr>
<td>NodeIndexContainsScan</td>
<td>Examines all values stored in an index, searching for entries containing a specific string.</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Leaf?</td>
<td>Updating?</td>
<td>Considerations</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>-------</td>
<td>-----------</td>
<td>-----------------</td>
</tr>
<tr>
<td>NodeIndexEndsWithScan</td>
<td>Examines all values stored in an index, searching for entries ending in a specific string.</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NodeIndexScan</td>
<td>Examines all values stored in an index, returning all nodes with a particular label having a specified property.</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NodeIndexSeek</td>
<td>Finds nodes using an index seek.</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NodeIndexSeekByRange</td>
<td>Finds nodes using an index seek where the value of the property matches the given prefix string.</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NodeLeftOuterHashJoin</td>
<td>Executes a left outer hash join.</td>
<td></td>
<td>Eager</td>
<td></td>
</tr>
<tr>
<td>NodeRightOuterHashJoin</td>
<td>Executes a right outer hash join.</td>
<td></td>
<td>Eager</td>
<td></td>
</tr>
<tr>
<td>NodeUniqueIndexSeek</td>
<td>Finds nodes using an index seek within a unique index.</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NodeUniqueIndexSeekByRange</td>
<td>Finds nodes using an index seek within a unique index where the value of the property matches the given prefix string.</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OrderedAggregation</td>
<td>Like EagerAggregation but relies on the ordering of incoming rows. Is not eager.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OrderedDistinct</td>
<td>Like Distinct but relies on the ordering of incoming rows.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Optional</td>
<td>Yields a single row with all columns set to null if no data is returned by its source.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Leaf?</td>
<td>Updating?</td>
<td>Considerations</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------</td>
<td>-----------</td>
<td>----------------</td>
</tr>
<tr>
<td>OptionalExpand(All)</td>
<td>Traverses relationships from a given node, producing a single row with the relationship and end node set to null if the predicates are not fulfilled.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OptionalExpand(Into)</td>
<td>Traverses all relationships between two nodes, producing a single row with the relationship and end node set to null if no matching relationships are found (the start node will be the node with the smallest degree).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PartialSort</td>
<td>Sorts a row by multiple columns if there is already an ordering.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PartialTop</td>
<td>Returns the first 'n' rows sorted by multiple columns if there is already an ordering.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ProcedureCall</td>
<td>Calls a procedure.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ProduceResults</td>
<td>Prepares the result so that it is consumable by the user.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ProjectEndpoints</td>
<td>Projects the start and end node of a relationship.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Projection</td>
<td>Evaluates a set of expressions, producing a row with the results thereof.</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RelationshipCountFrom</td>
<td>Uses the count store to answer questions about relationship counts.</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CountStore</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RemoveLabels</td>
<td>Deletes labels from a node.</td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>RollUpApply</td>
<td>Performs a nested loop. Executes a pattern expression or pattern comprehension.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Leaf?</td>
<td>Updating?</td>
<td>Considerations</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------</td>
<td>-----------</td>
<td>----------------</td>
</tr>
<tr>
<td>SelectOrAntiSemiApply</td>
<td>Performs a nested loop. Tests for the absence of a pattern predicate if an expression predicate evaluates to false.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SelectOrSemiApply</td>
<td>Performs a nested loop. Tests for the presence of a pattern predicate if an expression predicate evaluates to false.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SemiApply</td>
<td>Performs a nested loop. Tests for the presence of a pattern predicate.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SetLabels</td>
<td>Sets labels on a node.</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SetNodePropertiesFromMap</td>
<td>Sets properties from a map on a node.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SetProperty</td>
<td>Sets a property on a node or relationship.</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SetRelationshipPropertiesFromMap</td>
<td>Sets properties from a map on a relationship.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Skip</td>
<td>Skips 'n' rows from the incoming rows.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sort</td>
<td>Sorts rows by a provided key.</td>
<td></td>
<td>Eager</td>
<td></td>
</tr>
<tr>
<td>Top</td>
<td>Returns the first 'n' rows sorted by a provided key.</td>
<td></td>
<td>Eager</td>
<td></td>
</tr>
<tr>
<td>TriadicSelection</td>
<td>Solves triangular queries, such as the very common ‘find my friend-of-friends that are not already my friend’.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UndirectedRelationshipByIdSeek</td>
<td>Reads one or more relationships by id from the relationship store.</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Union</td>
<td>Concatenates the results from the right-hand operator with the results from the left-hand operator.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unwind</td>
<td>Returns one row per item in a list.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Leaf?</td>
<td>Updating?</td>
<td>Considerations</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------------------------------------------------------</td>
<td>-------</td>
<td>-----------</td>
<td>-----------------</td>
</tr>
<tr>
<td>ValueHashJoin</td>
<td>Executes a hash join on arbitrary values.</td>
<td></td>
<td></td>
<td>Eager</td>
</tr>
<tr>
<td>VarLengthExpand(All)</td>
<td>Traverses variable-length relationships from a given node.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VarLengthExpand(Into)</td>
<td>Finds all variable-length relationships between two nodes.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VarLengthExpand(Pruning)</td>
<td>Traverses variable-length relationships from a given node and only returns unique end nodes.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

14.3. Database hits (DbHits)

Each operator will send a request to the storage engine to do work such as retrieving or updating data. A database hit is an abstract unit of this storage engine work.

We list below all the actions that trigger one or more database hits:

- **Create actions**
  - Create a node
  - Create a relationship
  - Create a new node label
  - Create a new relationship type
  - Create a new ID for property keys with the same name

- **Delete actions**
  - Delete a node
  - Delete a relationship

- **Update actions**
  - Set one or more labels on a node
  - Remove one or more labels from a node

- **Node-specific actions**
  - Get a node by its ID
  - Get the degree of a node
  - Determine whether a node is dense
  - Determine whether a label is set on a node
  - Get the labels of a node
- Get a property of a node
- Get an existing node label
- Get the name of a label by its ID, or its ID by its name

- Relationship-specific actions
  - Get a relationship by its ID
  - Get a property of a relationship
  - Get an existing relationship type
  - Get a relationship type name by its ID, or its ID by its name

- General actions
  - Get the name of a property key by its ID, or its ID by the key name
  - Find a node or relationship through an index seek or index scan
  - Find a path in a variable-length expand
  - Find a shortest path
  - Ask the count store for a value

- Schema actions
  - Add an index
  - Drop an index
  - Get the reference of an index
  - Create a constraint
  - Drop a constraint

- Call a procedure
- Call a user-defined function

The presented value can vary slightly depending on the Cypher runtime that was used to execute the query. In the pipelined runtime the number of database hits will typically be higher since it uses a more accurate way of measuring.

14.4. Operators

All operators are listed here, grouped by the similarity of their characteristics.

14.4.1. All Nodes Scan

The AllNodesScan operator reads all nodes from the node store. The variable that will contain the nodes is seen in the arguments. Any query using this operator is likely to encounter performance problems on a non-trivial database.
14.4.2. Directed Relationship By Id Seek

The `DirectedRelationshipByIdSeek` operator reads one or more relationships by id from the relationship store, and produces both the relationship and the nodes on either side.
14.4.3. Node By Id Seek

The NodeByIdSeek operator reads one or more nodes by id from the node store.

Query

```
MATCH (n) WHERE id(n) = 0 RETURN n
```

Query Plan

```
Compiler CYPHER 4.0
Planner COST
Runtime PIPELINED
Runtime version 4.0

+-----------------+----------------+------+---------+-----------+
| Operator        | Estimated Rows | Rows | DB Hits | Variables |
| +ProduceResults |              1 |    1 |       0 | n         |
| +NodeByIdSeek   |              1 |    1 |       1 | n         |
+-----------------+----------------+------+---------+-----------+

Total database accesses: 1
```

14.4.4. Node By Label Scan

The NodeByLabelScan operator fetches all nodes with a specific label from the node label index.

Query

```
MATCH (person:Person) RETURN person
```

Query Plan

```
Compiler CYPHER 4.0
Planner COST
Runtime PIPELINED
Runtime version 4.0

+------------------+----------------+------+---------+---------+---------+
| Operator         | Estimated Rows | Rows | DB Hits | Variables | Other   |
| +ProduceResults  |             14 |   14 |       0 | person   |         |
| +NodeByLabelScan |             14 |   14 |      15 | person   | :Person |
+------------------+----------------+------+---------+---------+---------+

Total database accesses: 15
```

14.4.5. Node Index Seek

The NodeIndexSeek operator finds nodes using an index seek. The node variable and the index used is shown in the arguments of the operator. If the index is a unique index, the operator is instead called
NodeUniqueIndexSeek.

Query

```
MATCH (location:Location {name: 'Malmo'}) RETURN location
```

Query Plan

<table>
<thead>
<tr>
<th>Operator</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Order</th>
<th>Variables</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td></td>
<td>1</td>
<td></td>
<td>0</td>
<td>location</td>
<td></td>
</tr>
<tr>
<td>+NodeIndexSeek</td>
<td></td>
<td>1</td>
<td>2</td>
<td>location.name ASC</td>
<td>location</td>
<td>:Location(name)</td>
</tr>
</tbody>
</table>

Total database accesses: 2

14.4.6. Node Unique Index Seek

The **NodeUniqueIndexSeek** operator finds nodes using an index seek within a unique index. The node variable and the index used is shown in the arguments of the operator. If the index is not unique, the operator is instead called **NodeIndexSeek**. If the index seek is used to solve a **MERGE** clause, it will also be marked with (Locking). This makes it clear that any nodes returned from the index will be locked in order to prevent concurrent conflicting updates.

Query

```
MATCH (t:Team {name: 'Malmo'}) RETURN t
```

Query Plan

<table>
<thead>
<tr>
<th>Operator</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Order</th>
<th>Variables</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td></td>
<td>1</td>
<td></td>
<td>0</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>+NodeUniqueIndexSeek</td>
<td></td>
<td>1</td>
<td>1</td>
<td>t.name ASC</td>
<td>t</td>
<td>:Team(name)</td>
</tr>
</tbody>
</table>

Total database accesses: 1
14.4.7. Node Index Seek By Range

The `NodeIndexSeekByRange` operator finds nodes using an index seek where the value of the property matches a given prefix string. `NodeIndexSeekByRange` can be used for `STARTS WITH` and comparison operators such as `<`, `>`, `<=` and `>=`. If the index is a unique index, the operator is instead called `NodeUniqueIndexSeekByRange`.

Query

```
MATCH (l:Location) WHERE l.name STARTS WITH 'Lon' RETURN l
```

Query Plan

```
Compiler CYPHER 4.0
Planner COST
Runtime PIPELINED
Runtime version 4.0

<table>
<thead>
<tr>
<th>Operator</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Time (ms)</th>
<th>Order</th>
<th>Variables</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td></td>
<td>2</td>
<td></td>
<td>0</td>
<td>l.name ASC</td>
<td>l</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.217</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+NodeIndexSeekByRange</td>
<td></td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0.271</td>
<td>1.name ASC</td>
<td>1</td>
</tr>
<tr>
<td>:Location(name STARTS WITH $<code> AUTOSTRING0</code>)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total database accesses: 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

14.4.8. Node Unique Index Seek By Range

The `NodeUniqueIndexSeekByRange` operator finds nodes using an index seek within a unique index, where the value of the property matches a given prefix string. `NodeUniqueIndexSeekByRange` is used by `STARTS WITH` and comparison operators such as `<`, `>`, `<=` and `>=`. If the index is not unique, the operator is instead called `NodeIndexSeekByRange`.

Query

```
MATCH (t:Team) WHERE t.name STARTS WITH 'Ma' RETURN t
```

### 14.4.9. Node Index Contains Scan

The `NodeIndexContainsScan` operator examines all values stored in an index, searching for entries containing a specific string; for example, in queries including `CONTAINS`. Although this is slower than an index seek (since all entries need to be examined), it is still faster than the indirection resulting from a label scan using `NodeByLabelScan`, and a property store filter.

#### Query

```
MATCH (l:Location) WHERE l.name CONTAINS 'al' RETURN l
```

#### Query Plan

| Operator               | Estimated Rows | Rows | DB Hits | Order      | Variables | Other          |
|------------------------|----------------+------+---------+------------+-----------+----------------|
| +ProduceResults        | 0              | 2    | 0       | 1.name ASC | l         |                |
| +NodeIndexContainsScan | 0              | 2    | 3       | 1.name ASC | l         | :Location(name); "$` AUTOSTRING0` |

Total database accesses: 3
14.4.10. Node Index Ends With Scan

The NodeIndexEndsWithScan operator examines all values stored in an index, searching for entries ending in a specific string; for example, in queries containing ENDS WITH. Although this is slower than an index seek (since all entries need to be examined), it is still faster than the indirection resulting from a label scan using NodeByLabelScan, and a property store filter.

Query

```
MATCH (l:Location) WHERE l.name ENDS WITH 'al' RETURN l
```

Query Plan

```
Compiler CYPHER 4.0
Planner COST
Runtime PIPELINED
Runtime version 4.0

+------------------------+----------------+------+---------+------------+-----------
| Operator               | Estimated Rows | Rows | DB Hits | Order      | Variables | Other     |
+------------------------+----------------+------+---------+------------+-----------
| +ProduceResults        | 0              | 0    | 0       | l.name ASC | l         |
| |                      +----------------+------+---------+------------+-----------
| +NodeIndexEndsWithScan | 0              | 0    | 1       | l.name ASC | l         | :Location(name); $`AUTOSTRING0` |
+------------------------+----------------+------+---------+------------+-----------
Total database accesses: 1
```

14.4.11. Node Index Scan

The NodeIndexScan operator examines all values stored in an index, returning all nodes with a particular label having a specified property.

Query

```
MATCH (l:Location) WHERE exists(l.name) RETURN l
```
14.4.12. Undirected Relationship By Id Seek

The `UndirectedRelationshipByIdSeek` operator reads one or more relationships by id from the relationship store. As the direction is unspecified, two rows are produced for each relationship as a result of alternating the combination of the start and end node.

**Query**

```cypher
MATCH (n1)-[r]->()
WHERE id(r) = 1
RETURN r, n1
```

14.4.13. Apply

All the different `Apply` operators (listed below) share the same basic functionality: they perform a nested loop by taking a single row from the left-hand side, and using the `Argument` operator on the right-hand side, execute the operator tree on the right-hand side. The versions of the `Apply` operators differ in how the results are managed. The `Apply` operator (i.e. the standard version) takes the row produced by the right-hand side — which at this point contains data from both the left-hand and right-hand sides — and yields it. 
MATCH (p:Person {name: 'me'})
MATCH (q:Person {name: p.secondName})
RETURN p, q

14.4.14. Semi Apply

The SemiApply operator tests for the presence of a pattern predicate, and is a variation of the Apply operator. If the right-hand side operator yields at least one row, the row from the left-hand side operator is yielded by the SemiApply operator. This makes SemiApply a filtering operator, used mostly for pattern predicates in queries.

Query

MATCH (p:Person)
WHERE (p)-[:FRIENDS_WITH]->(:Person)
RETURN p.name
14.4.15. Anti Semi Apply

The *AntiSemiApply* operator tests for the absence of a pattern, and is a variation of the *Apply* operator. If the right-hand side operator yields no rows, the row from the left-hand side operator is yielded by the *AntiSemiApply* operator. This makes *AntiSemiApply* a filtering operator, used for pattern predicates in queries.

Query

```
MATCH (me:Person {name: "me"}), (other:Person)
WHERE NOT (me)-[:FRIENDS_WITH]->(other)
RETURN other.name
```
14.4.16. Let Semi Apply

The *LetSemiApply* operator tests for the presence of a pattern predicate, and is a variation of the *Apply* operator. When a query contains multiple pattern predicates separated with OR, *LetSemiApply* will be used to evaluate the first of these. It will record the result of evaluating the predicate but will leave any filtering to another operator. In the example, *LetSemiApply* will be used to check for the presence of the *FRIENDS_WITH* relationship from each person.

**Query**

```cypher
MATCH (other:Person)
WHERE (other)-[:FRIENDS_WITH]->(:Person) OR (other)-[:WORKS_IN]->(:Location)
RETURN other.name
```
14.4.17. Let Anti Semi Apply

The **LetAntiSemiApply** operator tests for the absence of a pattern, and is a variation of the **Apply** operator. When a query contains multiple negated pattern predicates — i.e. predicates separated with **OR**, where at least one predicate contains **NOT** — **LetAntiSemiApply** will be used to evaluate the first of these. It will
record the result of evaluating the predicate but will leave any filtering to another operator. In the example, LetAntiSemiApply will be used to check for the absence of the FRIENDS_WITH relationship from each person.

Query

```
MATCH (other:Person)
WHERE NOT ((other)-[:FRIENDS_WITH]-(Person)) OR (other)-[:WORKS_IN]-(Location)
RETURN other.name
```
14.4.18. Select Or Semi Apply

The SelectOrSemiApply operator tests for the presence of a pattern predicate and evaluates a predicate, and is a variation of the Apply operator. This operator allows for the mixing of normal predicates and pattern predicates that check for the presence of a pattern. First, the normal expression predicate is
evaluated, and, only if it returns false, is the costly pattern predicate evaluated.

Query

MATCH (other:Person)
WHERE other.age > 25 OR (other)-[:FRIENDS_WITH]->(:Person)
RETURN other.name

Query Plan

Compiler CYPHER 4.0
Planner COST
Runtime SLOTTED
Runtime version 4.0

<table>
<thead>
<tr>
<th>Operator</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Page Cache Hits</th>
<th>Page Cache Misses</th>
<th>Page Cache Hit Ratio</th>
<th>Variables</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>11</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>other, other.name</td>
<td></td>
</tr>
<tr>
<td>+Projection</td>
<td>11</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1.0000</td>
<td>other.name -- other</td>
<td>{other.name : cache[other.name]}</td>
</tr>
<tr>
<td>+SelectOrSemiApply</td>
<td>14</td>
<td>2</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>other</td>
<td>other.age &gt; $`AUTOINT0'</td>
</tr>
<tr>
<td>+CacheProperties</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>NODE71, REL53, other</td>
<td>cache[other.name]</td>
</tr>
<tr>
<td>+Filter</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1.0000</td>
<td>NODE71, REL53, other</td>
<td>NODE71:Person</td>
</tr>
<tr>
<td>+Expand(All)</td>
<td>2</td>
<td>2</td>
<td>16</td>
<td>15</td>
<td>0</td>
<td>1.0000</td>
<td>NODE71, REL53 -- other</td>
<td>(other)-[ REL53:FRIENDS_WITH]-&gt;( NODE71)</td>
</tr>
<tr>
<td>+Argument</td>
<td>14</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>other</td>
<td></td>
</tr>
<tr>
<td>+NodeByLabelScan</td>
<td>14</td>
<td>14</td>
<td>15</td>
<td>2</td>
<td>0</td>
<td>1.0000</td>
<td>other:Person</td>
<td></td>
</tr>
</tbody>
</table>

Total database accesses: 49

14.4.19. Select Or Anti Semi Apply

The SelectOrAntiSemiApply operator is used to evaluate OR between a predicate and a negative pattern predicate (i.e. a pattern predicate preceded with NOT), and is a variation of the Apply operator. If the predicate returns true, the pattern predicate is not tested. If the predicate returns false or null, SelectOrAntiSemiApply will instead test the pattern predicate.
14.4.20. Let Select Or Semi Apply

The `LetSelectOrSemiApply` operator is planned for pattern predicates that are combined with other predicates using `OR`. This is a variation of the `Apply` operator.

Query

```
MATCH (other:Person)
WHERE other.age > 25 OR NOT (other)-[:FRIENDS_WITH]->(Person)
RETURN other.name
```
### Query Plan

Compiler CYPHER 4.0  
Planner COST  
Runtime SLOTTED  
Runtime version 4.0

<table>
<thead>
<tr>
<th>Operator</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Page Cache Hits</th>
<th>Page Cache Misses</th>
<th>Page Cache Hit Ratio</th>
<th>Variables</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+NodeByLabelScan</td>
<td>1.0000</td>
<td></td>
<td>14</td>
<td>15</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Argument</td>
<td>0.0000</td>
<td>14</td>
<td>0</td>
<td>16</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Filter</td>
<td>1.0000</td>
<td>14</td>
<td>0</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Expand(All)</td>
<td>1.0000</td>
<td>15</td>
<td>12</td>
<td>24</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Argument</td>
<td>0.0000</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+LetSelectOrSemiApply</td>
<td>14</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Projection</td>
<td>1.36</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+SelectOrSemiApply</td>
<td>14</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+CacheProperties</td>
<td>2</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Filter</td>
<td>2</td>
<td></td>
<td>0</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Expand(All)</td>
<td>2</td>
<td></td>
<td>2</td>
<td>16</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Argument</td>
<td>14</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+NodeByLabelScan</td>
<td>14</td>
<td></td>
<td>15</td>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total database accesses: 97

---

### 14.4.21. Let Select Or Anti Semi Apply

The **LetSelectOrAntiSemiApply** operator is planned for negated pattern predicates — i.e. pattern predicates preceded with **NOT** — that are combined with other predicates using **OR**. This operator is a variation of the **Apply** operator.
Query

MATCH (other:Person)
WHERE NOT (other)-[:FRIENDS_WITH]->(:Person) OR (other)-[:WORKS_IN]->(:Location) OR other.age = 5
RETURN other.name

Query Plan

Compiler CYPHER 4.0
Planner COST
Runtime SLOTTED
Runtime version 4.0

Operator | Estimated Rows | Rows | DB Hits | Page Cache Hits | Page Cache Misses | Page Cost | Planner Cost
--- | --- | --- | --- | --- | --- | --- | ---
Operator | 11 | 14 | 0 | 0 | 0 | 0 | 0
+ProduceResults | 11 | 14 | 14 | 1 | 0 | 0 | 0
+Argument | 14 | 14 | 0 | 0 | 0 | 0 | 0
+Projection | 14 | 14 | 0 | 0 | 0 | 0 | 0
+SelectOrSemiApply | 14 | 14 | 0 | 0 | 0 | 0 | 0
+Filter | 15 | 0 | 2 | 0 | 0 | 0 | 0
+Expand(All) | 15 | 2 | 4 | 2 | 0 | 0 | 0
+Argument | 14 | 2 | 0 | 0 | 0 | 0 | 0
+LetSelectOrAntiSemiApply | 14 | 14 | 14 | 0 | 0 | 0 | 0
+CacheProperties | 2 | 0 | 0 | 0 | 0 | 0 | 0
+Filter | 2 | 0 | 2 | 1 | 0 | 0 | 0
+Expand(All) | 2 | 2 | 16 | 15 | 0 | 0 | 0
+Argument | 14 | 14 | 0 | 0 | 0 | 0 | 0
+NodeByLabelScan | 14 | 14 | 15 | 2 | 0 | 0 | 0
Total database accesses: 67
14.4.22. Conditional Apply

The **ConditionalApply** operator checks whether a variable is not null, and if so, the right child operator will be executed. This operator is a variation of the **Apply** operator.

**Query**

```cypher
MERGE (p:Person {name: 'Andy'})
ON MATCH SET p.exists = true
```

**Query Plan**

```
Compiler CYPHER 4.0
Planner COST
Runtime SLOTTED
Runtime version 4.0

<table>
<thead>
<tr>
<th>Operator</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Page Cache Hits</th>
<th>Page Cache Misses</th>
<th>Page Cache Hit Ratio</th>
<th>Order</th>
<th>Variables</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>0.0000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+----------------+------+---------+-----------------+-------------------+----------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+EmptyResult</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>0.0000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+----------------+------+---------+-----------------+-------------------+----------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+AntiConditionalApply</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>0.0000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+MergeCreateNode</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>0.0000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+----------------+------+---------+-----------------+-------------------+----------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ConditionalApply</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>0.0000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+----------------+------+---------+-----------------+-------------------+----------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+SetProperty</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>2.0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+----------------+------+---------+-----------------+-------------------+----------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+Argument</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>0.0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+----------------+------+---------+-----------------+-------------------+----------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+Optional</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>0.0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+----------------+------+---------+-----------------+-------------------+----------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+NodeIndexSeek</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>1.0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+----------------+------+---------+-----------------+-------------------+----------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Total database accesses: 5
```

14.4.23. Anti Conditional Apply

The **AntiConditionalApply** operator checks whether a variable is null, and if so, the right child operator
will be executed. This operator is a variation of the Apply operator.

Query

```cypher
MERGE (p:Person {name: 'Andy'})
ON CREATE SET p.exists = true
```

Query Plan

```cypher
Compiler CYPHER 4.0
Planner COST
Runtime SLOTTED
Runtime version 4.0
```

<table>
<thead>
<tr>
<th>Operator</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Page Cache Hits</th>
<th>Page Cache Misses</th>
<th>Page Cache Hit Ratio</th>
<th>Order</th>
<th>Variables</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+EmptyResult</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+AntiConditionalApply</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+SetProperty</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+MergeCreateNode</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Optional</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>p.name ASC</td>
<td></td>
</tr>
<tr>
<td>+NodeIndexSeek</td>
<td></td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td></td>
<td>p.name ASC</td>
<td>:Person(name)</td>
</tr>
</tbody>
</table>

Total database accesses: 2

14.4.24. Roll Up Apply

The RollUpApply operator is used to execute an expression which takes as input a pattern, and returns a list with content from the matched pattern; for example, when using a pattern expression or pattern comprehension in a query. This operator is a variation of the Apply operator.

Query

```cypher
MATCH (p:Person)
RETURN p.name, [(p)[::WORKS_IN]->(location) | location.name] AS cities
```
14.4.25. Argument

The **Argument** operator indicates the variable to be used as an argument to the right-hand side of an **Apply** operator.

**Query**

```
MATCH (s:Person {name: 'me'}) MERGE (s)-[:FRIENDS_WITH]->(s)
```

**Query Plan**

Compiler CYPHER 4.0
Planner COST
Runtime SLOTTED
Runtime version 4.0
14.4.26. Expand All

Given a start node, and depending on the pattern relationship, the Expand(All) operator will traverse incoming or outgoing relationships.
14.4.27. Expand Into

When both the start and end node have already been found, the Expand(Into) operator is used to find all relationships connecting the two nodes. As both the start and end node of the relationship are already in scope, the node with the smallest degree will be used. This can make a noticeable difference when dense nodes appear as end points.
### Query Plan

Compiler CYPHER 4.0
Planner COST
Runtime PIPELINED
Runtime version 4.0

<table>
<thead>
<tr>
<th>Operator</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Time (ms)</th>
<th>Order</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.000</td>
<td>p.name ASC</td>
<td>anon[30], anon[53], fof, p</td>
</tr>
<tr>
<td>+Filter</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.000</td>
<td>p.name ASC</td>
<td>anon[30], anon[53], fof, p</td>
</tr>
<tr>
<td>+Expand(Into)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.008</td>
<td>p.name ASC</td>
<td>(p)-[:FRIENDS_WITH]-&gt;(fof)</td>
</tr>
<tr>
<td>+Expand(All)</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td></td>
<td>p.name ASC</td>
<td>(p)&lt;-(fof)</td>
</tr>
<tr>
<td>+NodeIndexSeek</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
<td>p.name ASC</td>
<td>:Person(name)</td>
</tr>
</tbody>
</table>

Total database accesses: 5

### 14.4.28. Optional Expand All

The **OptionalExpand(All)** operator is analogous to **Expand(All)**, apart from when no relationships match the direction, type and property predicates. In this situation, **OptionalExpand(all)** will return a single row with the relationship and end node set to **null**.

**Query**

```cypher
MATCH (p:Person)
OPTIONAL MATCH (p)-[works_in:WORKS_IN]->(l) WHERE works_in.duration > 180
RETURN p, l
```
14.4.29. Optional Expand Into

The \texttt{OptionalExpand(Into)} operator is analogous to \texttt{Expand(Into)}, apart from when no matching relationships are found. In this situation, \texttt{OptionalExpand(Into)} will return a single row with the relationship and end node set to \texttt{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.

\begin{verbatim}
MATCH (p:Person)-[works_in:WORKS_IN]->(l)
  OPTIONAL MATCH (l)<-[:WORKS_IN]-(p)
RETURN p
\end{verbatim}
14.4.30. VarLength Expand All

Given a start node, the `VarLengthExpand(All)` operator will traverse variable-length relationships.

Query

```
MATCH (p:Person)-[:FRIENDS_WITH *1..2]-(:Person) RETURN p, q
```
14.4.31. VarLength Expand Into

When both the start and end node have already been found, the VarLengthExpand(Into) operator is used to find all variable-length relationships connecting the two nodes.

Query

MATCH (p:Person)-[:FRIENDS_WITH *1..2]-(:Person) RETURN p
14.4.32. VarLength Expand Pruning

Given a start node, the `VarLengthExpand(Pruning)` operator will traverse variable-length relationships much like the `VarLengthExpand(All)` operator. However, as an optimization, some paths will not be explored if they are guaranteed to produce an end node that has already been found (by means of a previous path traversal). This will only be used in cases where the individual paths are not of interest. This operator guarantees that all the end nodes produced will be unique.

Query

```cypher
MATCH (p:Person)-[:FRIENDS_WITH*3..4]-(q:Person) RETURN DISTINCT p, q
```

Query Plan

<table>
<thead>
<tr>
<th>Operator</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Time (ms)</th>
<th>Variables</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.010</td>
<td>p, q</td>
<td></td>
</tr>
<tr>
<td>+Distinct</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.019</td>
<td>p, q</td>
<td>p, q</td>
</tr>
<tr>
<td>+Filter</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.076</td>
<td>p, q</td>
<td>q:Person</td>
</tr>
<tr>
<td>+VarLengthExpand(Pruning)</td>
<td>0</td>
<td>0</td>
<td>32</td>
<td>10.816</td>
<td>q -- p</td>
<td>(p)- [:FRIENDS_WITH*3..4]-(q)</td>
</tr>
<tr>
<td>+NodeByLabelScan</td>
<td>14</td>
<td>14</td>
<td>15</td>
<td>0.086</td>
<td>p</td>
<td>:Person</td>
</tr>
</tbody>
</table>

Total database accesses: 47

14.4.33. Assert Same Node

The `AssertSameNode` operator is used to ensure that no unique constraints are violated. The example looks for the presence of a team with the supplied name and id, and if one does not exist, it will be created. Owing to the existence of two unique constraints on `:Team(name)` and `:Team(id)`, any node that would be found by the `UniqueIndexSeek` must be the very same node, or the constraints would be violated.

Query

```cypher
MERGE (t:Team {name: 'Engineering', id: 42})
```
14.4.34. Drop Result

The DropResult operator produces zero rows. It is applied when it can be deduced through static analysis that the result of an expression will be empty, such as when a predicate guaranteed to return false (e.g. 1 > 5) is used in a query.

Query

MATCH (p) WHERE false RETURN p
14.4.35. Empty Result

The **EmptyResult** operator eagerly loads all incoming data and discards it.

Query

```
CREATE (:Person)
```

Query Plan

```
Compiler CYPHER 4.0
Planner COST
Runtime SLOTTED
Runtime version 4.0

<table>
<thead>
<tr>
<th>Operator</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Page Cache Hits</th>
<th>Page Cache Misses</th>
<th>Page Cache Hit Ratio</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>anon[8]</td>
</tr>
<tr>
<td>+EmptyResult</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>anon[8]</td>
</tr>
<tr>
<td>+Create</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>anon[8]</td>
</tr>
</tbody>
</table>

Total database accesses: 1
```
14.4.36. Produce Results

The ProduceResults operator prepares the result so that it is consumable by the user, such as transforming internal values to user values. It is present in every single query that returns data to the user, and has little bearing on performance optimisation.

Query

```cypher
MATCH (n)
RETURN n
```

Query Plan

```
<table>
<thead>
<tr>
<th>Operator</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>35</td>
<td>35</td>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>+AllNodesScan</td>
<td>35</td>
<td>35</td>
<td>36</td>
<td>n</td>
</tr>
</tbody>
</table>
```

Total database accesses: 36

14.4.37. Load CSV

The LoadCSV operator loads data from a CSV source into the query. It is used whenever the LOAD CSV clause is used in a query.

Query

```cypher
LOAD CSV FROM 'https://neo4j.com/docs/cypher-refcard/3.3/csv/artists.csv' AS line
RETURN line
```
14.4.38. Hash joins in general

Hash joins have two inputs: the build input and probe input. The query planner assigns these roles so that the smaller of the two inputs is the build input. The build input is pulled in eagerly, and is used to build a probe table. Once this is complete, the probe table is checked for each row coming from the probe input side.

In query plans, the build input is always the left operator, and the probe input the right operator.

There are four hash join operators:

- NodeHashJoin
- ValueHashJoin
- NodeLeftOuterHashJoin
- NodeRightOuterHashJoin

14.4.39. Node Hash Join

The NodeHashJoin operator is a variation of the hash join. NodeHashJoin executes the hash join on node ids. As primitive types and arrays can be used, it can be done very efficiently.

Query

```
MATCH (bob:Person {name:'Bob'})-[[:WORKS_IN]]->(loc)<-[[:WORKS_IN]]-(matt:Person {name:'Mattis'})
RETURN loc.name
```
### 14.4.40. Value Hash Join

The `ValueHashJoin` operator is a variation of the `hash join`. This operator allows for arbitrary values to be used as the join key. It is most frequently used to solve predicates of the form \( n.prop1 = m.prop2 \) (i.e. equality predicates between two property columns).

**Query**

```
MATCH (p:Person), (q:Person)
WHERE p.age = q.age
RETURN p, q
```
14.4.41. Node Left/Right Outer Hash Join

The NodeLeftOuterHashJoin and NodeRightOuterHashJoin operators are variations of the hash join. The query below can be planned with either a left or a right outer join. The decision depends on the cardinalities of the left-hand and right-hand sides; i.e. how many rows would be returned, respectively, for \((a:Person)\) and \((a)\rightarrow(b:Person)\). If \((a:Person)\) returns fewer results than \((a)\rightarrow(b:Person)\), a left outer join — indicated by NodeLeftOuterHashJoin — is planned. On the other hand, if \((a:Person)\) returns more results than \((a)\rightarrow(b:Person)\), a right outer join — indicated by NodeRightOuterHashJoin — is planned instead.

Query

```cypher
MATCH (a:Person)
OPTIONAL MATCH (a)--> (b:Person)
USING JOIN ON a
RETURN a.name, b.name
```
### 14.4.42. Triadic Selection

The **TriadicSelection** operator is used to solve triangular queries, such as the very common 'find my friend-of-friends that are not already my friend'. It does so by putting all the friends into a set, and uses the set to check if the friend-of-friends are already connected to me. The example finds the names of all friends of my friends that are not already my friends.

**Query**

```cypher
MATCH (me:Person)-[:FRIENDS_WITH]-()-[:FRIENDS_WITH]-{other}
WHERE NOT (me)-[:FRIENDS_WITH]-{other}
RETURN other.name
```
14.4.43. Cartesian Product

The **CartesianProduct** operator produces a cartesian product of the two inputs — each row coming from the left child operator will be combined with all the rows from the right child operator. **CartesianProduct** generally exhibits bad performance and ought to be avoided if possible.

Query

```
MATCH (p:Person), (t:Team) RETURN p, t
```
**Query Plan**

Compiler CYPHER 4.0
Planner COST
Runtime PIPELINED
Runtime version 4.0

<table>
<thead>
<tr>
<th>Operator</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Time (ms)</th>
<th>Variables</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>140</td>
<td>140</td>
<td>0</td>
<td>5.204</td>
<td>p, t</td>
<td></td>
</tr>
<tr>
<td>+CartesianProduct</td>
<td>140</td>
<td>140</td>
<td>0</td>
<td>0.787</td>
<td>t -- p</td>
<td></td>
</tr>
<tr>
<td>NodeByLabelScan</td>
<td>14</td>
<td>14</td>
<td>15</td>
<td>0.125</td>
<td>p</td>
<td>:Person</td>
</tr>
<tr>
<td>NodeByLabelScan</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td>0.118</td>
<td>t</td>
<td>:Team</td>
</tr>
</tbody>
</table>

Total database accesses: 26

### 14.4.44. Foreach

The **Foreach** operator executes a nested loop between the left child operator and the right child operator. In an analogous manner to the **Apply** operator, it takes a row from the left-hand side and, using the **Argument** operator, provides it to the operator tree on the right-hand side. **Foreach** will yield all the rows coming in from the left-hand side; all results from the right-hand side are pulled in and discarded.

**Query**

```cypher
FOREACH (value IN [1, 2, 3] |
CREATE (:Person {age: value}))
```
14.4.45. Eager

For isolation purposes, the **Eager** operator ensures that operations affecting subsequent operations are executed fully for the whole dataset before continuing execution. Information from the stores is fetched in a lazy manner; i.e. the pattern matching might not be fully exhausted before updates are applied. To guarantee reasonable semantics, the query planner will insert **Eager** operators into the query plan to prevent updates from influencing pattern matching; this scenario is exemplified by the query below, where the **DELETE** clause influences the **MATCH** clause. The **Eager** operator can cause high memory usage when importing data or migrating graph structures. In such cases, the operations should be split into simpler steps; e.g. importing nodes and relationships separately. Alternatively, the records to be updated can be returned, followed by an update statement.

**Query**

```
MATCH (a)-[r]->(b) DELETE r, a, b MERGE()
```

### Query Plan

**Compiler**: CYPHER 4.0  
**Planner**: COST  
**Runtime**: SLOTTED  
**Runtime version**: 4.0

<table>
<thead>
<tr>
<th>Operator</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Page Cache Hits</th>
<th>Page Cache Misses</th>
<th>Page Cache Hit Ratio</th>
<th>Variables Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>anon[38], a, b, r</td>
</tr>
<tr>
<td>+EmptyResult</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>anon[38], a, b, r</td>
</tr>
<tr>
<td>+Apply</td>
<td>1</td>
<td>504</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>a, b, r -- anon[38]</td>
</tr>
<tr>
<td>+AntiConditionalApply</td>
<td>1</td>
<td>504</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>anon[38]</td>
</tr>
<tr>
<td>+MergeCreateNode</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>anon[38]</td>
</tr>
<tr>
<td>+Optional</td>
<td>35</td>
<td>504</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>anon[38]</td>
</tr>
<tr>
<td>+AllNodesScan</td>
<td>35</td>
<td>504</td>
<td>540</td>
<td>0</td>
<td>0</td>
<td>1.0000</td>
<td>b, r -- a</td>
</tr>
<tr>
<td>+Eager</td>
<td>36</td>
<td>36</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>a, b, r</td>
</tr>
<tr>
<td>+Delete(3)</td>
<td>36</td>
<td>36</td>
<td>39</td>
<td>28</td>
<td>0</td>
<td>2.0000</td>
<td>a, b, r</td>
</tr>
<tr>
<td>+Eager</td>
<td>36</td>
<td>36</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>a, b, r</td>
</tr>
<tr>
<td>+Expand(All)</td>
<td>36</td>
<td>36</td>
<td>71</td>
<td>22</td>
<td>0</td>
<td>1.0000</td>
<td>b, r -- a</td>
</tr>
<tr>
<td>+AllNodesScan</td>
<td>35</td>
<td>35</td>
<td>36</td>
<td>1</td>
<td>0</td>
<td>1.0000</td>
<td>a</td>
</tr>
</tbody>
</table>

**Total database accesses**: 686

### 14.4.46. Eager Aggregation

The **EagerAggregation** operator evaluates a grouping expression and uses the result to group rows into different groupings. For each of these groupings, **EagerAggregation** will then evaluate all aggregation functions and return the result. To do this, **EagerAggregation**, as the name implies, needs to pull in all data.
eagerly from its source and build up state, which leads to increased memory pressure in the system.

**Query**

```
MATCH (l:Location)<-[:WORKS_IN]-(p:Person) RETURN l.name AS location, collect(p.name) AS people
```

**Query Plan**

```
Compiler CYPHER 4.0
Planer COST
Runtime PIPELINED
Runtime version 4.0

| Operator          | Estimated Rows | Rows | DB Hits | Time (ms) | Variables        | Other |
|-------------------+----------------+------+---------+-----------+------------------|
| +ProduceResults   | 4              | 6    | 0       | 0.563     | location, people |
| +EagerAggregation | 4              | 6    | 30      | 0.188     | location, people | location |
| +Filter           | 15             | 15   | 15      |           | anon[19], l, p   | p:Person |
| +Expand(All)      | 15             | 15   | 26      |           | anon[19], p -- l | (l)<-[:WORKS_IN]-(p) |
| +CacheProperties  | 10             | 10   | 20      |           | l                | cache[l.name] |
| +NodeByLabelScan  | 10             | 10   | 11      |           | l                | :Location |

Total database accesses: 102

14.4.47. Ordered Aggregation

The **OrderedAggregation** operator is an optimization of the **EagerAggregation** operator that takes advantage of the ordering of the incoming rows. This operator uses lazy evaluation and has a lower memory pressure in the system than the **EagerAggregation** operator.

**Query**

```
MATCH (p:Person) WHERE p.name STARTS WITH 'P' RETURN p.name, count(*) AS count
```
### 14.4.48. Node Count From Count Store

The `NodeCountFromCountStore` operator uses the count store to answer questions about node counts. This is much faster than the `EagerAggregation` operator which achieves the same result by actually counting. However, as the count store only stores a limited range of combinations, `EagerAggregation` will still be used for more complex queries. For example, we can get counts for all nodes, and nodes with a label, but not nodes with more than one label.

**Query**

```cypher
MATCH (p:Person) RETURN count(p) AS people
```
### 14.4.49. Relationship Count From Count Store

The `RelationshipCountFromCountStore` operator uses the count store to answer questions about relationship counts. This is much faster than the `EagerAggregation` operator which achieves the same result by actually counting. However, as the count store only stores a limited range of combinations, `EagerAggregation` will still be used for more complex queries. For example, we can get counts for all relationships, relationships with a type, relationships with a label on one end, but not relationships with labels on both end nodes.

**Query**

```cypher
MATCH (p:Person)-[r:WORKS_IN]->() RETURN count(r) AS jobs
```

**Query Plan**

```
<table>
<thead>
<tr>
<th>Operator</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Variables</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>people</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+RelationshipCountFromCountStore</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>people</td>
<td>count( (:Some(Person)) AS jobs people )</td>
</tr>
</tbody>
</table>

Total database accesses: 1
```
14.4.50. Distinct

The Distinct operator removes duplicate rows from the incoming stream of rows. To ensure only distinct elements are returned, Distinct will pull in data lazily from its source and build up state. This may lead to increased memory pressure in the system.

Query

```cypher
MATCH (l:Location)<-[[:WORKS_IN]]-(p:Person) RETURN DISTINCT l
```

Query Plan

Compiler CYPHER 4.0
Planner COST
Runtime PIPELINED
Runtime version 4.0

```
+------------------+----------------+------+---------+-----------+------------------+
| Operator         | Estimated Rows | Rows | DB Hits | Time (ms) | Variables        | Other             |
+------------------+----------------+------+---------+-----------+------------------+
| +ProduceResults  |             14 |    6 |       0 |     0.250 | l                |
|                  +----------------+------+---------+-----------+------------------+
| +Distinct        |             14 |    6 |       0 |     1.476 | l                | l                |
|                  +----------------+------+---------+-----------+------------------+
| +Filter          |             15 |   15 |      15 |           | anon[19], l, p   | p:Person          |
|                  +----------------+------+---------+-----------+------------------+
| +Expand(All)     |             15 |   15 |      26 |           | anon[19], p -- l | (l)<-[[:WORKS_IN]]-(p) |
|                  +----------------+------+---------+-----------+------------------+
| +NodeByLabelScan |             10 |   10 |      11 |           | l                | :Location         |

Total database accesses: 52
```

14.4.51. Ordered Distinct

The OrderedDistinct operator is an optimization of the Distinct operator that takes advantage of the ordering of the incoming rows. This operator has a lower memory pressure in the system than the Distinct operator.

Query

```cypher
MATCH (p:Person) WHERE p.name STARTS WITH 'P' RETURN DISTINCT p.name
```
14.4.52. Filter

The Filter operator filters each row coming from the child operator, only passing through rows that evaluate the predicates to true.

Query

```cypher
MATCH (p:Person) WHERE p.name =~ '^a.*' RETURN p
```

Query Plan

```plaintext
<table>
<thead>
<tr>
<th>Operator</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Variables</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>p</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+Filter</td>
<td></td>
<td></td>
<td>1</td>
<td>cache[p.name] =~ $<code>  AUTOSTRING0</code></td>
</tr>
<tr>
<td></td>
<td>+NodeIndexScan</td>
<td>14</td>
<td>14</td>
<td>15</td>
<td>:Person(name) = $<code>  AUTOSTRING0</code>, cache[p.name]</td>
</tr>
</tbody>
</table>
```

Total database accesses: 15
14.4.53. Limit

The **Limit** operator returns the first 'n' rows from the incoming input.

**Query**

```
MATCH (p:Person) RETURN p LIMIT 3
```

**Query Plan**

Compiler CYPHER 4.0
Planner COST
Runtime PIPELINED
Runtime version 4.0

```
<table>
<thead>
<tr>
<th>Operator</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Time (ms)</th>
<th>Variables</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td></td>
<td>3</td>
<td>3</td>
<td>0.286</td>
<td>p</td>
<td></td>
</tr>
<tr>
<td>+Limit</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>1.525</td>
<td>p</td>
<td>3</td>
</tr>
<tr>
<td>+NodeByLabelScan</td>
<td>14</td>
<td>14</td>
<td>15</td>
<td>0.164</td>
<td>p</td>
<td>:Person</td>
</tr>
</tbody>
</table>
```

Total database accesses: 15

14.4.54. Skip

The **Skip** operator skips 'n' rows from the incoming rows.

**Query**

```
MATCH (p:Person)
RETURN p
ORDER BY p.id
SKIP 1
```
14.4.55. Sort

The **Sort** operator sorts rows by a provided key. In order to sort the data, all data from the source operator needs to be pulled in eagerly and kept in the query state, which will lead to increased memory pressure in the system.

**Query**

```cypher
MATCH (p:Person) RETURN p ORDER BY p.name
```
14.4.56. Partial Sort

The PartialSort operator is an optimization of the Sort operator that takes advantage of the ordering of the incoming rows. This operator uses lazy evaluation and has a lower memory pressure in the system than the Sort operator. Partial sort is only applicable when sorting on multiple columns.

Query

```
MATCH (p:Person) WHERE p.name STARTS WITH 'P' RETURN p ORDER BY p.name, p.age
```
### Query Plan

<table>
<thead>
<tr>
<th>Operator</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Page Cache Hits</th>
<th>Page Cache Misses</th>
<th>Page Cache Hit Ratio</th>
<th>Order</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1.0000</td>
<td>p.name ASC, p.age ASC</td>
<td>p, p.age, p.name</td>
</tr>
<tr>
<td>+PartialSort</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>p.name ASC, p.age ASC</td>
<td>p, p.age, p.name</td>
</tr>
<tr>
<td>+Projection</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>p.name ASC</td>
<td>p.age, p.name -- p</td>
</tr>
<tr>
<td>+NodeIndexSeekByRange</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0.0000</td>
<td>p.name ASC</td>
<td>:Person(name STARTS WITH $' AUTOSTRING0'), cache[p.name]</td>
</tr>
</tbody>
</table>

Total database accesses: 5

### 14.4.57. Top

The **Top** operator returns the first 'n' rows sorted by a provided key. Instead of sorting the entire input, only the top 'n' rows are retained.

**Query**

```
MATCH (p:Person) RETURN p ORDER BY p.name LIMIT 2
```
**Query Plan**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Time (ms)</th>
<th>Order</th>
<th>Variables</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0.289</td>
<td>p.name ASC</td>
<td>p, p.name</td>
<td></td>
</tr>
<tr>
<td>+Top</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1.466</td>
<td>p.name ASC</td>
<td>p, p.name</td>
<td>p.name; 2</td>
</tr>
<tr>
<td>+Projection</td>
<td>14</td>
<td>14</td>
<td>28</td>
<td></td>
<td>p.name -- p</td>
<td>(p.name: p.name)</td>
<td></td>
</tr>
<tr>
<td>+NodeByLabelScan</td>
<td>14</td>
<td>14</td>
<td>15</td>
<td></td>
<td>p</td>
<td>:Person</td>
<td></td>
</tr>
</tbody>
</table>

Total database accesses: 43

**14.4.58. Partial Top**

The **PartialTop** operator is an optimization of the **Top** operator that takes advantage of the ordering of the incoming rows. This operator uses lazy evaluation and has a lower memory pressure in the system than the **Top** operator. Partial top is only applicable when sorting on multiple columns.

**Query**

MATCH (p:Person) WHERE p.name STARTS WITH 'P' RETURN p ORDER BY p.name, p.age LIMIT 2
### Query Plan

Compiler CYPHER 4.0  
Planner COST  
Runtime SLOTTED  
Runtime version 4.0

| Operator              | Estimated Rows | Rows | DB Hits | Page Cache Hits | Page Cache Misses | Page Cache Hit Ratio | Order                 | Variables          | Other |
|-----------------------+----------------+------+---------+-----------------+-----------------+----------------------+-----------------------+-------------------+-------|
| +ProduceResults       |              0 |    2 |       0 |               2 |                 0 | 1.0000 | p.name ASC, p.age ASC | p, p.age, p.name   | p.name ASC, p.age ASC | p, p.age, p.name   |
| +PartialTop           |              0 |    2 |       0 |               0 |                 0 | 0.0000 | p.name ASC, p.age ASC | p, p.age, p.name   | p.name; p.age; 2   |
| +Projection           |              0 |    2 |       2 |               0 |                 0 | 0.0000 | p.name ASC            | p.age, p.name -- p | {p.name : cache[p.name], p.age : p.age} |
| +NodeIndexSeekByRange |              0 |    2 |       3 |               0 |                 1 | 0.0000 | p.name ASC            | p                  | :Person(name STARTS WITH $` AUTOSTRING0`), cache[p.name] |

Total database accesses: 5

### 14.4.59. Union

The **Union** operator concatenates the results from the right child operator with the results from the left child operator.

**Query**

```
MATCH (p:Location)  
RETURN p.name  
UNION ALL  
MATCH (p:Country)  
RETURN p.name
```
14.4.60. Unwind

The **Unwind** operator returns one row per item in a list.

**Query**

```cypher
UNWIND range(1, 5) as value return value
```
14.4.61. Lock Nodes

The **LockNodes** operator locks the start and end node when creating a relationship.

**Query**

```cypher
MATCH (s:Person {name: 'me'})
MERGE (s)-[:FRIENDS_WITH]->(s)
```

**Query Plan**

```
Compiler CYPHER 4.0
Planner COST
Runtime SLOTTED
Runtime version 4.0

<table>
<thead>
<tr>
<th>Operator</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Page Cache Hits</th>
<th>Page Cache Misses</th>
<th>Page Cache Hit Ratio</th>
<th>Order</th>
<th>Variables</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>anon[40], s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+EmptyResult</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>anon[40], s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Apply</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>anon[40], s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+AntiConditionalApply</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>anon[40], s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+MergeCreateRelationship</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1.0000</td>
<td>anon[40] -- s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+Argument</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+AntiConditionalApply</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>anon[40], s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
14.4.62. Optional

The Optional operator is used to solve some OPTIONAL MATCH queries. It will pull data from its source, simply passing it through if any data exists. However, if no data is returned by its source, Optional will yield a single row with all columns set to null.

Query

```
MATCH (p:Person {name: 'me'}) OPTIONAL MATCH (q:Person {name: 'Lulu'}) RETURN p, q
```
14.4.63. Project Endpoints

The ProjectEndpoints operator projects the start and end node of a relationship.

Query

```
CREATE (n)-[p:KNOWS]->(m) WITH p AS r MATCH (u)-[r]->(v) RETURN u, v
```
14.4.64. Projection

For each incoming row, the Projection operator evaluates a set of expressions and produces a row with the results of the expressions.

Query

```
RETURN 'hello' AS greeting
```
14.4.65. Empty Row

The `EmptyRow` operator returns a single row with no columns.

Query

```cypher
FOREACH (value IN [1, 2, 3] |
    CREATE (:Person {age: value})
)
```
Query Plan

Compiler CYPHER 4.0
Planner COST
Runtime SLOTTED
Runtime version 4.0

+-----------------+----------------+------+---------+-----------------+-------------------+
| Operator        | Estimated Rows | Rows | DB Hits | Page Cache Hits | Page Cache Misses | Page Cache Hit Ratio | Variables         |
+-----------------+----------------+------+---------+-----------------+-------------------+--------------------------+--------------------|
| +ProduceResults |              1 |    0 |       0 |               0 |                 0 | 0.0000 |                   |
|                 |               +----------------+------+---------+-----------------+-------------------+--------------------------+--------------------|
| +EmptyResult    |              1 |    0 |       0 |               0 |                 0 | 0.0000 |                   |
|                 |               +----------------+------+---------+-----------------+-------------------+--------------------------+--------------------|
| +Foreach        |              1 |    1 |       0 |               0 |                 0 | 0.0000 |                   |
| | +Create       |              1 |    3 |       9 |               0 |                 0 | 0.0000 | anon[36] -- value |
| | | Argument     |              1 |    3 |       0 |               0 |                 0 | 0.0000 | value             |
| | |               |               +----------------+------+---------+-----------------+-------------------+--------------------------+--------------------|
| +EmptyRow       |              1 |    1 |       0 |               0 |                 0 | 0.0000 |                   |
+-----------------+----------------+------+---------+-----------------+-------------------+-------------------+--------------------|

Total database accesses: 9

14.4.66. Procedure Call

The ProcedureCall operator indicates an invocation to a procedure.

Query

    CALL db.labels() YIELD label RETURN * ORDER BY label
14.4.67. Cache Properties

The CacheProperties operator reads nodes and relationship properties and caches them in the current row. Future accesses to these properties can avoid reading from the store which will speed up the query. In the plan below we will cache `l.name` before `Expand(All)` where there are fewer rows.

Query

```
MATCH (l:Location)<-[::WORKS_IN]-(p:Person) RETURN l.name AS location, p.name AS name
```
14.4.68. Create Nodes / Relationships

The **Create** operator is used to create nodes and relationships.

**Query**

```
CREATE (max:Person {name: 'Max'}), (chris:Person {name: 'Chris'})
CREATE (max)->[:FRIENDS_WITH]->(chris)
```
14.4.69. Delete

The **Delete** operator is used to delete a node or a relationship.

**Query**

```
MATCH (me:Person {name: 'me'})-[w:WORKS_IN {duration: 190}]->(london:Location {name: 'London'})
DELETE w
```
14.4.70. Detach Delete

The \texttt{DetachDelete} operator is used in all queries containing the \texttt{DETACH DELETE} clause, when deleting nodes and their relationships.

\textbf{Query}

\texttt{MATCH (p:Person) DETACH DELETE p}
14.4.71. Merge Create Node

The MergeCreateNode operator is used when creating a node as a result of a MERGE clause failing to find the node.

Query

```
MERGE (:Person {name: 'Sally'})
```
14.4.72. Merge Create Relationship

The `MergeCreateRelationship` operator is used when creating a relationship as a result of a `MERGE` clause failing to find the relationship.

Query

```
MATCH (s:Person {name: 'Sally'})
MERGE (s)-[:FRIENDS_WITH]->(s)
```

Query Plan

```
Compiler CYPHER 4.0
Planner COST
Runtime SLOTTED
Runtime version 4.0
```

14.4.73. Set Labels

The **SetLabels** operator is used when setting labels on a node.

**Query**

```sql
MATCH (n)
SET n:Person
```

Total database accesses: 1
### Query Plan

Compiler CYPHER 4.0  
Planner COST  
Runtime SLOTTED  
Runtime version 4.0

| Operator        | Estimated Rows | Rows | DB Hits | Page Cache Hits | Page Cache Misses | Page Cache Hit Ratio | Variables |
|-----------------|----------------+------|---------|-----------------|-------------------|---------------------|------------|
| +ProduceResults | 35             | 0    | 0        | 0               | 0                 | 0                   | n         |
| +EmptyResult    | 35             | 0    | 0        | 0               | 0                 | 0                   | n         |
| +SetLabels      | 35             | 35   | 35       | 2               | 0                 | 1                   | n         |
| +AllNodesScan   | 35             | 35   | 36       | 1               | 0                 | 1                   | n         |

Total database accesses: 71

### 14.4.74. Remove Labels

The `REMOVE` operator is used when deleting labels from a node.

**Query**

```cypher
MATCH (n)
REMOVE n:Person
```
Query Plan

Compiler CYPHER 4.0
Planner COST
Runtime SLOTTED
Runtime version 4.0

<table>
<thead>
<tr>
<th>Operator</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Page Cache Hits</th>
<th>Page Cache Misses</th>
<th>Page Cache Hit Ratio</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>35</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>+EmptyResult</td>
<td>35</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>+RemoveLabels</td>
<td>35</td>
<td>35</td>
<td>35</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>n</td>
</tr>
<tr>
<td>+AllNodesScan</td>
<td>35</td>
<td>35</td>
<td>36</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>n</td>
</tr>
</tbody>
</table>

Total database accesses: 71

14.4.75. Set Node Properties From Map

The SetNodePropertiesFromMap operator is used when setting properties from a map on a node.

Query

```
MATCH (n)
SET n = {weekday: 'Monday', meal: 'Lunch'}
```
### Query Plan

<table>
<thead>
<tr>
<th>Operator</th>
<th>Estimated Rows</th>
<th>Rows</th>
<th>DB Hits</th>
<th>Page Cache Hits</th>
<th>Page Cache Misses</th>
<th>Page Cache Hit Ratio</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ProduceResults</td>
<td>35</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>n</td>
</tr>
<tr>
<td>+EmptyResult</td>
<td>35</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>n</td>
</tr>
<tr>
<td>+SetNodePropertiesFromMap</td>
<td>35</td>
<td>35</td>
<td>141</td>
<td>3</td>
<td>0</td>
<td>1.0000</td>
<td>n</td>
</tr>
<tr>
<td>+AllNodesScan</td>
<td>35</td>
<td>35</td>
<td>36</td>
<td>1</td>
<td>0</td>
<td>1.0000</td>
<td></td>
</tr>
</tbody>
</table>

Total database accesses: 177

### 14.4.76. Set Relationship Properties From Map

The `SetRelationshipPropertiesFromMap` operator is used when setting properties from a map on a relationship.

**Query**

```
MATCH (n)-[r]->(m)
SET r = {weight: 5, unit: 'kg'}
```
Query Plan

Compiler CYpher 4.0
Planner COST
Runtime SLOTTED
Runtime version 4.0

| Operator                          | Estimated Rows | Rows | DB Hits | Page Cache Hits | Page Cache Misses | Page Cache Hit Ratio | Variables | Other         |
|-----------------------------------|----------------+------+---------+----------------+------------------|-------------------|-----------------|---------------------|
| +ProduceResults                   | 18             | 0     | 0        | 0              | 0                | 0.0000             | m, n, r    |               |
| |                                   +----------------+------+---------+----------------+------------------|-------------------|-----------------|                     |
| +EmptyResult                      | 18             | 0     | 0        | 0              | 0                | 0.0000             | m, n, r    |               |
| |                                   +----------------+------+---------+----------------+------------------|-------------------|-----------------|                     |
| +SetRelationshipPropertiesFromMap | 18             | 18    | 69       | 0              | 0                | 0.0000             | m, n, r    |               |
| |                                   +----------------+------+---------+----------------+------------------|-------------------|-----------------|                     |
| +Expand(All)                      | 18             | 18    | 53       | 22             |                  | 1.0000             | n, r -- m  | (m)<-[r:]->(n) |
| |                                   +----------------+------+---------+----------------+------------------|-------------------|-----------------|                     |
| +AllNodesScan                     | 35             | 35    | 36       | 1              |                  | 1.0000             | m           |               |
| |                                   +----------------+------+---------+----------------+------------------|-------------------|-----------------|                     |

Total database accesses: 158

14.4.77. Set Property

The `setProperty` operator is used when setting a property on a node or relationship.

Query

```
MATCH (n)
SET n.checked = true
```
14.4.78. Create Unique Constraint

The `CreateUniqueConstraint` operator creates a unique constraint on a property for all nodes having a certain label. The following query will create a unique constraint with the name `uniqueness` on the `name` property of nodes with the `Country` label.

**Query**

```
CREATE CONSTRAINT uniqueness ON (c:Country) ASSERT c.name is UNIQUE
```

**Query Plan**

```
<table>
<thead>
<tr>
<th>Operator</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>+CreateUniquePropertyConstraint</td>
<td>uniqueness</td>
</tr>
</tbody>
</table>
```

Total database accesses: 74
14.4.79. Drop Unique Constraint ==\textcolor{red}{\textbf{Deprecated}}

The \texttt{DropUniqueConstraint} operator removes a unique constraint from a property for all nodes having a certain label. The following query will drop a unique constraint on the \texttt{name} property of nodes with the \texttt{Country} label.

Query

\begin{verbatim}
DROP CONSTRAINT ON (c:Country) ASSERT c.name is UNIQUE
\end{verbatim}

Query Plan

Compiler CYPHER 4.0
Planner ADMINISTRATION
Runtime SCHEMA
Runtime version 4.0

\begin{verbatim}
+-------------------------------+
| Operator                      |
+-------------------------------+
| +DropUniquePropertyConstraint |
+-------------------------------+

Total database accesses: ?

14.4.80. Create Node Property Existence Constraint

The \texttt{CreateNodePropertyExistenceConstraint} operator creates an existence constraint with the name \texttt{existence} on a property for all nodes having a certain label. This will only appear in Enterprise Edition.

Query

\begin{verbatim}
CREATE CONSTRAINT existence ON (p:Person) ASSERT exists(p.name)
\end{verbatim}

Query Plan

Compiler CYPHER 4.0
Planner ADMINISTRATION
Runtime SCHEMA
Runtime version 4.0

\begin{verbatim}
+----------------------------------------+-----------+
| Operator                               | Other     |
+----------------------------------------+-----------+
| +CreateNodePropertyExistenceConstraint | existence |
+----------------------------------------+-----------+

Total database accesses: ?
14.4.81. Drop Node Property Existence Constraint == Deprecated

The `DropNodePropertyExistenceConstraint` operator removes an existence constraint from a property for all nodes having a certain label. This will only appear in Enterprise Edition.

Query

```
DROP CONSTRAINT ON (p:Person) ASSERT exists(p.name)
```

Query Plan

```
Compiler CYPHER 4.0
Planner ADMINISTRATION
Runtime SCHEMA
Runtime version 4.0
+--------------------------------------+
| Operator                             |
+--------------------------------------+
| +DropNodePropertyExistenceConstraint |
+--------------------------------------+
Total database accesses: ?
```

14.4.82. Create Node Key Constraint

The `CreateNodeKeyConstraint` operator creates a node key constraint with the name `node_key` which ensures that all nodes with a particular label have a set of defined properties whose combined value is unique, and where all properties in the set are present. This will only appear in Enterprise Edition.

Query

```
CREATE CONSTRAINT node_key ON (e:Employee) ASSERT (e.firstname, e.surname) IS NODE KEY
```

Query Plan

```
Compiler CYPHER 4.0
Planner ADMINISTRATION
Runtime SCHEMA
Runtime version 4.0
+--------------------------+----------+
| Operator                 | Other    |
+--------------------------+----------+
| +CreateNodeKeyConstraint | node_key |
+--------------------------+----------+
Total database accesses: ?
```

14.4.83. Drop Node Key Constraint == Deprecated

The `DropNodeKeyConstraint` operator removes a node key constraint from a set of properties for all nodes having a certain label. This will only appear in Enterprise Edition.
Query

DROP CONSTRAINT ON (e:Employee) ASSERT (e.firstname, e.surname) IS NODE KEY

Query Plan

Compiler CYPHER 4.0
Planner ADMINISTRATION
Runtime SCHEMA
Runtime version 4.0

Total database accesses: ?

14.4.84. Create Relationship Property Existence Constraint

The CreateRelationshipPropertyExistenceConstraint operator creates an existence constraint with the name existence on a property for all relationships of a certain type. This will only appear in Enterprise Edition.

Query

CREATE CONSTRAINT existence ON ()-[l:LIKED]-() ASSERT exists(l.when)

Query Plan

Compiler CYPHER 4.0
Planner ADMINISTRATION
Runtime SCHEMA
Runtime version 4.0

Total database accesses: ?

14.4.85. Drop Relationship Property Existence Constraint ==

Deprecated

The DropRelationshipPropertyExistenceConstraint operator removes an existence constraint from a property for all relationships of a certain type. This will only appear in Enterprise Edition.
Query

```cypher
DROP CONSTRAINT ON ()-[l:LIKED]-() ASSERT exists(l.when)
```

Query Plan

```
Compiler CYPHER 4.0
Planner ADMINISTRATION
Runtime SCHEMA
Runtime version 4.0
+----------------------------------------------+
| Operator                                     |
| +DropRelationshipPropertyExistenceConstraint |
+----------------------------------------------+
Total database accesses: ?
```

14.4.86. Drop Constraint by name

The `DropConstraint` operator removes a constraint using the name of the constraint, no matter the type.

Query

```cypher
DROP CONSTRAINT name
```

Query Plan

```
Compiler CYPHER 4.0
Planner ADMINISTRATION
Runtime SCHEMA
Runtime version 4.0
+-----------------+-------+
| Operator        | Other |
| +DropConstraint | name  |
+-----------------+-------+
Total database accesses: ?
```

14.4.87. Create Index

The `CreateIndex` operator creates an index on a property for all nodes having a certain label. The following query will create an index with the name `my_index` on the `name` property of nodes with the `Country` label.

Query

```cypher
CREATE INDEX my_index FOR (c:Country) ON (c.name)
```
14.4.88. Drop Index == Deprecated

The DropIndex operator removes an index from a property for all nodes having a certain label. The following query will drop an index on the name property of nodes with the Country label.

Query

```
DROP INDEX ON :Country(name)
```

Query Plan

Compiler CYPHER 4.0
Planner ADMINISTRATION
Runtime SCHEMA
Runtime version 4.0

```
+-----------------+
| Operator         |
| +CreateIndex     |
| my_index         |
+-----------------+
```

Total database accesses: ?

14.4.89. Drop Index by name

The DropIndex operator removes an index using the name of the index.

Query

```
DROP INDEX name
```
14.5. 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.
14.5.1. Shortest path with fast algorithm

Query

```
MATCH (KevinB:Person {name: 'Kevin Bacon'}),
(Al:Person {name: 'Al Pacino'}),
p = shortestPath((KevinB)-[:ACTED_IN*]-(Al))
WHERE all(r IN relationships(p) WHERE exists(r.role))
RETURN p
```

This query can be evaluated with the fast algorithm — there are no predicates that need to see the whole path before being evaluated.

Query plan

```
Compiler CYPHER 4.0
Planner COST
Runtime SLOTTED
Runtime version 4.0
```

| Operator       | Estimated Rows | Rows | DB Hits | Page Cache Hits | Page Cache Misses | Page Cache Hit Ratio | Order | Variables                  | Other |
|----------------+----------------+------+---------+-----------------+-------------------+---------------------+-------+----------------------------+-------|
| +ProduceResults | 2              | 1    | 0       | 0               | 0                 | 0.0000              | KevinB.name ASC | anon[117], Al, KevinB, p |
|                |                |      |         |                 |                   |                     |                   |                     |       |
| +ShortestPath   | 2              | 1    | 13      | 23              | 0                 | 1.0000              | KevinB.name ASC | anon[117], p -- Al, KevinB | {p0 : all(r IN relationships(p) WHERE exists(r.role))} |
|                |                |      |         |                 |                   |                     |                   |                     |       |
| +CartesianProduct | 2              | 1    | 0       | 0               | 0                 | 0.0000              | KevinB.name ASC | KevinB -- Al |
|                |                |      |         |                 |                   |                     |                   |                     |       |
| +NodeIndexSeek  | 1              | 1    | 2       | 1               | 0                 | 1.0000              | Al.name ASC      | Al                   | :Person(name) |
|                |                |      |         |                 |                   |                     |                   |                     |       |
| +NodeIndexSeek  | 1              | 1    | 2       | 0               | 1                 | 0.0000              | KevinB.name ASC | KevinB | :Person(name) |
|                |                |      |         |                 |                   |                     |                   |                     |       |
```

Total database accesses: 17

14.5.2. Shortest path with additional predicate checks on the paths
Consider using the exhaustive search as a fallback

Predicates used in the \texttt{WHERE} clause that apply to the shortest path pattern are evaluated before deciding what the shortest matching path is.

\textbf{Query}

\begin{verbatim}
MATCH (KevinB:Person {name: 'Kevin Bacon'}),
   (Al:Person {name: 'Al Pacino'}),
p = shortestPath((KevinB)-[*]-(Al))
WHERE length(p) > 1
RETURN p
\end{verbatim}

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.

\textbf{Query plan}

\begin{verbatim}
Compiler CYPHER 4.0
Planner COST
Runtime SLOTTED
Runtime version 4.0

+--------------------------+----------------+------+---------+-----------------+-------------------+
| Operator                 | Estimated Rows | Rows | DB Hits | Page Cache Hits | Page Cache Misses | Page |
|                          |                |      |         |                |                  | Cache |
|                          |                |      |         |                |                  | Hit Ratio |
|                          |                |      |         |                |                  | Order      |
|                          |                |      |         |                |                  | Variables  |
|                          |                |      |         |                |                  | Other      |
| +--------------------------+----------------+------+---------+-----------------+-------------------+
| +ProduceResults          |              1 |    1 |       0 |               0 |                 0 | 0.0000 |
|                          | KevinB.name ASC| anon[94], anon[116], Al, KevinB, p  |
|                          +----------------+------+---------+-----------------+-------------------+
| +AntiConditionalApply    |              1 |    1 |       0 |               0 |                 0 | 0.0000 |
|                          | KevinB.name ASC| anon[94], anon[116], Al, KevinB, p  |
|                          +----------------+------+---------+-----------------+-------------------+
| | +Top                   |              0 |    0 |       0 |               0 |                 0 | 0.0000 |
|                          |                 | anon[94], anon[116], Al, KevinB, p  |
|                          |                 |            |             | length(p) > $`  AUTOINT2` |
|                          +----------------+------+---------+-----------------+-------------------+

545
\end{verbatim}
|   | +Projection          | 0 | 0 | 0 | 0 | 0 | 0 |
|   | PathExpression(NodePathStep(Variable(KevinB), MultiRelationshipPathStep(Variable(anon[116]), BOTH, Some(Variable(KevinB))), NilPathStep())) |   |     |
| +Apply                      | 0 | 0 | 0 | 0 | 0 | 0 |
|   | | +Argument              | 2 | 0 | 0 | 0 | 0 | 0 |
|   |   | | +Optional              | 1 | 1 | 0 | 0 | 0 | 0 |
|   |   | | +ShortestPath           | 1 | 1 | 1 | 21 | 0 | 0 |
|   |   | | +Argument              | 2 | 1 | 0 | 0 | 0 | 0 |
|   |   | | +CartesianProduct       | 2 | 1 | 0 | 0 | 0 | 0 |
|   |   | | +NodeIndexSeek          | 1 | 1 | 2 | 1 | 0 | 0 |
|   |   | | +NodeIndexSeek          | 1 | 1 | 2 | 1 | 0 | 0 |

Total database accesses: 5
The way the bigger exhaustive query plan works is by using `Apply/Optional` to ensure that when the fast algorithm does not find any results, a `null` result is generated instead of simply stopping the result stream. On top of this, the planner will issue an `AntiConditionalApply`, which will run the exhaustive search if the path variable is pointing to `null` instead of a path.

An `ErrorPlan` operator will appear in the execution plan in cases where (i) `cypher.forbid_exhaustive_shortestpath` is set to `true`, and (ii) the fast algorithm is not able to find the shortest path.

Prevent the exhaustive search from being used as a fallback

**Query**

```cypher
MATCH (KevinB:Person {name: 'Kevin Bacon'}),
     (Al:Person {name: 'Al Pacino'}),
     p = shortestPath((KevinB)-[*]-(Al))
WITH p
WHERE length(p) > 1
RETURN p
```

This query, just like the one above, needs to check that the whole path follows the predicate before we know if it is valid or not. However, the inclusion of the `WITH` clause means that the query plan will not include the fallback to the slower exhaustive search algorithm. Instead, any paths found by the fast algorithm will subsequently be filtered, which may result in no answers being returned.
Query plan

Compiler CYPHER 4.0

Planner COST

Runtime SLOTTED

Runtime version 4.0

Total database accesses: 5
Chapter 15. Deprecations, additions and compatibility

Cypher is a language that is constantly evolving. New features get added to the language continuously, and occasionally, some features become deprecated and are subsequently removed.

This section list 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.

15.1. Version 4.0

15.1.1. Removed features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Removed rels()</td>
<td>Replaced by relationships().</td>
</tr>
<tr>
<td>Function Removed toInt()</td>
<td>Replaced by toInteger().</td>
</tr>
<tr>
<td>Function Removed lower()</td>
<td>Replaced by toLower().</td>
</tr>
<tr>
<td>Function Removed upper()</td>
<td>Replaced by toUpper().</td>
</tr>
<tr>
<td>Function Removed extract()</td>
<td>Replaced by list comprehension.</td>
</tr>
<tr>
<td>Feature</td>
<td>Details</td>
</tr>
<tr>
<td>-------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Function Removed</strong></td>
<td><strong>removed</strong></td>
</tr>
<tr>
<td></td>
<td>Replaced by list comprehension.</td>
</tr>
<tr>
<td></td>
<td><strong>filter()</strong></td>
</tr>
<tr>
<td><strong>Functionality Removed</strong></td>
<td><strong>removed</strong></td>
</tr>
<tr>
<td>For Rule planner:</td>
<td>The <code>RULE</code> planner was removed in 3.2, but still possible to trigger using <code>START</code> or <code>CREATE UNIQUE</code> clauses. Now it is completely removed.</td>
</tr>
<tr>
<td></td>
<td><strong>CYPHER planner=rule</strong></td>
</tr>
<tr>
<td><strong>Functionality Removed</strong></td>
<td><strong>removed</strong></td>
</tr>
<tr>
<td>Explicit indexes</td>
<td>The removal of the <code>RULE</code> planner in 3.2 was the beginning of the end for explicit indexes. Now they are completely removed, including the removal of the built-in procedures for Neo4j 3.3 to 3.5.</td>
</tr>
<tr>
<td><strong>Functionality Removed</strong></td>
<td><strong>removed</strong></td>
</tr>
<tr>
<td>For compiled runtime:</td>
<td>Replaced by the new pipelined runtime which covers a much wider range of queries.</td>
</tr>
<tr>
<td></td>
<td><strong>CYPHER runtime=compiled</strong></td>
</tr>
<tr>
<td><strong>Clause Removed</strong></td>
<td><strong>removed</strong></td>
</tr>
<tr>
<td><strong>CREATE UNIQUE</strong></td>
<td>Running queries with this clause will cause a syntax error. Running with CYPHER 3.5 will cause a runtime error due to the removal of the rule planner.</td>
</tr>
<tr>
<td><strong>Clause Removed</strong></td>
<td><strong>removed</strong></td>
</tr>
<tr>
<td><strong>START</strong></td>
<td>Running queries with this clause will cause a syntax error. Running with CYPHER 3.5 will cause a runtime error due to the removal of the rule planner.</td>
</tr>
<tr>
<td><strong>Syntax Removed</strong></td>
<td><strong>removed</strong></td>
</tr>
<tr>
<td>MATCH (n)-[:A</td>
<td>B</td>
</tr>
<tr>
<td><strong>Syntax Removed</strong></td>
<td><strong>removed</strong></td>
</tr>
<tr>
<td>MATCH (n)-[x:A</td>
<td>B</td>
</tr>
<tr>
<td><strong>Syntax Removed</strong></td>
<td><strong>removed</strong></td>
</tr>
<tr>
<td>MATCH (n)-[x:A</td>
<td>B</td>
</tr>
</tbody>
</table>
### 15.1.2. Deprecated features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td>Removed</td>
</tr>
<tr>
<td><code>{parameter}</code></td>
<td>Replaced by <code>$parameter</code>.</td>
</tr>
</tbody>
</table>

### As in Cypher 3.2, this is replaced by:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td>Deprecated</td>
</tr>
<tr>
<td><code>MATCH (n)-[rs*]-() RETURN rs</code></td>
<td>Replaced by <code>MATCH p=(n)-[*]-() RETURN relationships(p) AS rs</code>.</td>
</tr>
<tr>
<td><strong>Syntax</strong></td>
<td>Deprecated</td>
</tr>
<tr>
<td><code>CREATE INDEX ON :Label(prop)</code></td>
<td>Replaced by <code>CREATE INDEX FOR (n:Label) ON (n.prop)</code>.</td>
</tr>
<tr>
<td><strong>Syntax</strong></td>
<td>Deprecated</td>
</tr>
<tr>
<td><code>DROP INDEX ON :Label(prop)</code></td>
<td>Replaced by <code>DROP INDEX name</code>.</td>
</tr>
<tr>
<td><strong>Syntax</strong></td>
<td>Deprecated</td>
</tr>
<tr>
<td><code>DROP CONSTRAINT ON (n:Label) ASSERT (n.prop) IS NODE KEY</code></td>
<td>Replaced by <code>DROP CONSTRAINT name</code>.</td>
</tr>
<tr>
<td><strong>Syntax</strong></td>
<td>Deprecated</td>
</tr>
<tr>
<td><code>DROP CONSTRAINT ON (n:Label) ASSERT (n.prop) IS UNIQUE</code></td>
<td>Replaced by <code>DROP CONSTRAINT name</code>.</td>
</tr>
<tr>
<td><strong>Syntax</strong></td>
<td>Deprecated</td>
</tr>
<tr>
<td><code>DROP CONSTRAINT ON ()-[r:Type]-() ASSERT exists(r.prop)</code></td>
<td>Replaced by <code>DROP CONSTRAINT name</code>.</td>
</tr>
</tbody>
</table>
### 15.1.3. Restricted features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function</strong></td>
<td><strong>Restricted</strong></td>
</tr>
<tr>
<td><code>length()</code></td>
<td>Restricted to only work on paths. See <code>length()</code> for more details.</td>
</tr>
<tr>
<td><strong>Function</strong></td>
<td><strong>Restricted</strong></td>
</tr>
<tr>
<td><code>size()</code></td>
<td>No longer works for paths. Only works for strings, lists and pattern expressions. See <code>size()</code> for more details.</td>
</tr>
</tbody>
</table>

### 15.1.4. Updated features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td><strong>Extended</strong></td>
</tr>
<tr>
<td><code>CREATE CONSTRAINT [name] ON ...</code></td>
<td>The create constraint syntax can now include a name.</td>
</tr>
</tbody>
</table>

### 15.1.5. New features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Functionality</strong></td>
<td><strong>New</strong></td>
</tr>
<tr>
<td>Pipelined runtime:</td>
<td>This Neo4j Enterprise Edition only feature involves a new runtime that has many performance enhancements.</td>
</tr>
<tr>
<td><code>CYpher runtime=pipelined</code></td>
<td></td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td><strong>New</strong></td>
</tr>
<tr>
<td>Multi-database administration</td>
<td>New Cypher commands for administering multiple databases.</td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td><strong>New</strong></td>
</tr>
<tr>
<td>Security administration</td>
<td>New Cypher commands for administering role-based access-control.</td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td><strong>New</strong></td>
</tr>
<tr>
<td>Fine-grained security</td>
<td>New Cypher commands for administering dbms, database, graph and sub-graph access control.</td>
</tr>
<tr>
<td><strong>Syntax</strong></td>
<td><strong>New</strong></td>
</tr>
<tr>
<td><code>CREATE INDEX [name] FOR (n:Label) ON (n.prop)</code></td>
<td>New syntax for creating indexes, which can include a name.</td>
</tr>
</tbody>
</table>
### 15.2. Version 3.5

#### 15.2.1. Deprecated features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function</strong> Deprecated</td>
<td><strong>extract()</strong> Replaced by list comprehension.</td>
</tr>
<tr>
<td><strong>Function</strong> Deprecated</td>
<td><strong>filter()</strong> Replaced by list comprehension.</td>
</tr>
</tbody>
</table>
| **Functionality** Deprecated | Compiled runtime: 
**CYPHER runtime=compiled** 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. |

### 15.2.2. Version 3.4
<table>
<thead>
<tr>
<th>Feature</th>
<th>Type</th>
<th>Change</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spatial point types</td>
<td>Functionality</td>
<td>Amendment</td>
<td>A point — irrespective of which Coordinate Reference System is used — can be stored as a property and is able to be backed by an index. Prior to this, a point was a virtual property only.</td>
</tr>
<tr>
<td>point() - Cartesian 3D</td>
<td>Function</td>
<td>Added</td>
<td></td>
</tr>
<tr>
<td>point() - WGS 84 3D</td>
<td>Function</td>
<td>Added</td>
<td></td>
</tr>
<tr>
<td>randomUUID()</td>
<td>Function</td>
<td>Added</td>
<td></td>
</tr>
<tr>
<td>Temporal types</td>
<td>Functionality</td>
<td>Added</td>
<td>Supports storing, indexing and working with the following temporal types: Date, Time, LocalTime, DateTime, LocalDateTime and Duration.</td>
</tr>
<tr>
<td>Temporal functions</td>
<td>Functionality</td>
<td>Added</td>
<td>Functions allowing for the creation and manipulation of values for each temporal type — Date, Time, LocalTime, DateTime, LocalDateTime and Duration.</td>
</tr>
<tr>
<td>Temporal operators</td>
<td>Functionality</td>
<td>Added</td>
<td>Operators allowing for the manipulation of values for each temporal type — Date, Time, LocalTime, DateTime, LocalDateTime and Duration.</td>
</tr>
<tr>
<td>toString()</td>
<td>Function</td>
<td>Extended</td>
<td>Now also allows temporal values as input (i.e. values of type Date, Time, LocalTime, DateTime, LocalDateTime or Duration).</td>
</tr>
</tbody>
</table>

15.2.3. Version 3.3
### Version 3.2

<table>
<thead>
<tr>
<th>Feature</th>
<th>Type</th>
<th>Change</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>START</strong> Clause</td>
<td>Removed</td>
<td>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></td>
<td></td>
</tr>
<tr>
<td><strong>CYPHER runtime=slotted</strong> (Faster interpreted runtime)</td>
<td>Functionality</td>
<td>Added</td>
<td>Neo4j Enterprise Edition only</td>
</tr>
<tr>
<td><code>max()</code>, <code>min()</code></td>
<td>Function</td>
<td>Extended</td>
<td>Now also supports aggregation over sets containing lists of strings and/or numbers, as well as over sets containing strings, numbers, and lists of strings and/or numbers</td>
</tr>
</tbody>
</table>

#### 15.2.4. Version 3.2

<table>
<thead>
<tr>
<th>Feature</th>
<th>Type</th>
<th>Change</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CYPHER planner=rule</strong> (Rule planner)</td>
<td>Functionality</td>
<td>Removed</td>
<td>All queries now use the cost planner. Any query prepended thus will fall back to using Cypher 3.1.</td>
</tr>
<tr>
<td><strong>CREATE UNIQUE</strong> Clause</td>
<td>Removed</td>
<td>Running such queries will fall back to using Cypher 3.1 (and use the rule planner)</td>
<td></td>
</tr>
<tr>
<td><strong>START</strong> Clause</td>
<td>Removed</td>
<td>Running such queries will fall back to using Cypher 3.1 (and use the rule planner)</td>
<td></td>
</tr>
<tr>
<td><em><em>MATCH (n)-[rs</em>]-() RETURN rs</em>* Syntax</td>
<td>Deprecated</td>
<td>Replaced by <code>MATCH p=(n)-[*]-() RETURN relationships(p)</code> AS rs</td>
<td></td>
</tr>
<tr>
<td><strong>MATCH (n)-[:A]:B[:C {foo: 'bar'}]-() RETURN n</strong> Syntax</td>
<td>Deprecated</td>
<td>Replaced by <code>MATCH (n)-[:A]:B[:C {foo: 'bar'}]-() RETURN n</code></td>
<td></td>
</tr>
<tr>
<td><strong>MATCH (n)-[x:A]:B[:C]-() RETURN n</strong> Syntax</td>
<td>Deprecated</td>
<td>Replaced by <code>MATCH (n)-[x:A]:B[:C]-() RETURN n</code></td>
<td></td>
</tr>
<tr>
<td>Feature</td>
<td>Type</td>
<td>Change</td>
<td>Details</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
<td>--------</td>
<td>---------</td>
</tr>
<tr>
<td>MATCH (n)-[x:A</td>
<td>B</td>
<td>C*]-() RETURN n</td>
<td>Syntax</td>
</tr>
<tr>
<td>User-defined aggregation functions</td>
<td>Functionality</td>
<td>Added</td>
<td></td>
</tr>
<tr>
<td>Composite indexes</td>
<td>Index</td>
<td>Added</td>
<td></td>
</tr>
<tr>
<td>Node Key</td>
<td>Index</td>
<td>Added</td>
<td>Neo4j Enterprise Edition only</td>
</tr>
<tr>
<td>CYPHER runtime=compiled (Compiled runtime)</td>
<td>Functionality</td>
<td>Added</td>
<td>Neo4j Enterprise Edition only</td>
</tr>
<tr>
<td>reverse()</td>
<td>Function</td>
<td>Extended</td>
<td>Now also allows a list as input</td>
</tr>
<tr>
<td>max(), min()</td>
<td>Function</td>
<td>Extended</td>
<td>Now also supports aggregation over a set containing both strings and numbers</td>
</tr>
</tbody>
</table>

15.2.5. Version 3.1

<table>
<thead>
<tr>
<th>Feature</th>
<th>Type</th>
<th>Change</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>rels()</td>
<td>Function</td>
<td>Deprecated</td>
<td>Replaced by relationships()</td>
</tr>
<tr>
<td>toInt()</td>
<td>Function</td>
<td>Deprecated</td>
<td>Replaced by toInteger()</td>
</tr>
<tr>
<td>lower()</td>
<td>Function</td>
<td>Deprecated</td>
<td>Replaced by toLower()</td>
</tr>
<tr>
<td>upper()</td>
<td>Function</td>
<td>Deprecated</td>
<td>Replaced by toUpper()</td>
</tr>
<tr>
<td>toBoolean()</td>
<td>Function</td>
<td>Added</td>
<td></td>
</tr>
<tr>
<td>Map projection</td>
<td>Syntax</td>
<td>Added</td>
<td></td>
</tr>
<tr>
<td>Pattern comprehension</td>
<td>Syntax</td>
<td>Added</td>
<td></td>
</tr>
<tr>
<td>User-defined functions</td>
<td>Functionality</td>
<td>Added</td>
<td></td>
</tr>
<tr>
<td>CALL...YIELD...WHERE</td>
<td>Clause</td>
<td>Extended</td>
<td>Records returned by YIELD may be filtered further using WHERE</td>
</tr>
</tbody>
</table>

15.2.6. Version 3.0

<table>
<thead>
<tr>
<th>Feature</th>
<th>Type</th>
<th>Change</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>has()</td>
<td>Function</td>
<td>Removed</td>
<td>Replaced by exists()</td>
</tr>
<tr>
<td>str()</td>
<td>Function</td>
<td>Removed</td>
<td>Replaced by toString()</td>
</tr>
<tr>
<td>(parameter)</td>
<td>Syntax</td>
<td>Deprecated</td>
<td>Replaced by $parameter</td>
</tr>
<tr>
<td>properties()</td>
<td>Function</td>
<td>Added</td>
<td></td>
</tr>
<tr>
<td>CALL [...YIELD]</td>
<td>Clause</td>
<td>Added</td>
<td></td>
</tr>
</tbody>
</table>
The ability of Neo4j to support multiple older versions of the Cypher language has been changing. In versions of Neo4j before 3.5 the backwards compatibility layer included the Cypher language parser, planner and runtime. All supported versions of Cypher would run on the same Neo4j kernel. In Neo4j 3.4, however, this was changed such that the compatibility layer no longer included the runtime. This meant that running, for example, a `CYPHER 3.1` query inside Neo4j 3.5 would plan the query using the 3.1 planner, and run it using the 3.5 runtime and kernel. In Neo4j 4.0 this was changed again, such that the compatibility layer includes only the parser. For example, running a `CYPHER 3.5` query inside Neo4j will parse older language features, but plan using the 4.0 planner, and run using the 4.0 runtime and kernel. The primary reason for this change has been optimizations in the Cypher runtime to allow Cypher query to perform better.

Older versions of the language can still be accessed if required. There are two ways to select which version to use in queries.

1. Setting a version for all queries: You can configure your database with the configuration parameter `cypher.default_language_version`, and enter which version you’d like to use (see Supported language versions). Every Cypher query will use this version, provided the query hasn’t explicitly been configured as described in the next item below.

2. Setting a version on a query by query basis: The other method is to set the version for a particular query. Prepending a query with `CYPHER 3.5` will execute the query with the version of Cypher included in Neo4j 3.5.

Below is an example using the older parameter syntax `{param}`:

```
CYPHER 3.5
MATCH (n:Person)
WHERE n.age > {agelimit}
RETURN n.name, n.age
```

Without the `CYPHER 3.5` prefix this query would fail with a syntax error. With `CYPHER 3.5` however, it will only generate a warning and still work.
In Neo4j 4.0 some older language features are understood by the Cypher parser even if they are no longer supported by the Neo4j kernel. These features will result in runtime errors. See the table at Cypher Version 4.0 for the list of affected features.

15.4. Supported language versions

Neo4j 4.0 supports the following versions of the Cypher language:

- Neo4j Cypher 3.5
- Neo4j Cypher 4.0

Each release of Neo4j supports a limited number of old Cypher Language Versions. When you upgrade to a new release of Neo4j, please make sure that it supports the Cypher language version you need. If not, you may need to modify your queries to work with a newer Cypher language version.
Chapter 16. 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

16.1. Clauses

<table>
<thead>
<tr>
<th>Clause</th>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL [...] YIELD]</td>
<td>Reading/Writing</td>
<td>Invoke a procedure deployed in the database.</td>
</tr>
<tr>
<td>CALL [...]</td>
<td>Reading/Writing</td>
<td>Evaluates a subquery, typically used for post-union processing or aggregations.</td>
</tr>
<tr>
<td>CREATE</td>
<td>Writing</td>
<td>Create nodes and relationships.</td>
</tr>
<tr>
<td>CREATE CONSTRAINT [existence] ON (n:Label) ASSERT exists(n.property)]</td>
<td>Schema</td>
<td>Create a constraint ensuring that all nodes with a particular label have a certain property.</td>
</tr>
<tr>
<td>CREATE CONSTRAINT [node_key] ON (n:Label) ASSERT (n.prop1, ... , n.propN) IS NODE KEY]</td>
<td>Schema</td>
<td>Create a constraint ensuring all nodes with a particular label have all the specified properties and that the combination of property values is unique; i.e. ensures existence and uniqueness.</td>
</tr>
<tr>
<td>CREATE CONSTRAINT [existence] ON ()-<a href="">r:REL_TYPE</a>-] ASSERT exists(r.property)]</td>
<td>Schema</td>
<td>Create a constraint ensuring that all relationship with a particular type have a certain property.</td>
</tr>
<tr>
<td>CREATE CONSTRAINT [uniqueness] ON (n:Label) ASSERT n.property IS UNIQUE]</td>
<td>Schema</td>
<td>Create a constraint ensuring the uniqueness of the combination of node label and property value for a particular property key across all nodes.</td>
</tr>
<tr>
<td>CREATE INDEX [single] FOR (n:Label) ON (n.property)]</td>
<td>Schema</td>
<td>Create an index on all nodes with a particular label and a single property; i.e. create a single-property index.</td>
</tr>
<tr>
<td>Clause</td>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>CREATE INDEX [composite] FOR (n:Label) ON (n.prop1, ..., n.propN)]</td>
<td>Schema</td>
<td>Create an index on all nodes with a particular label and multiple properties; i.e. create a composite index.</td>
</tr>
<tr>
<td>DELETE</td>
<td>Writing</td>
<td>Delete nodes, relationships or paths. Any node to be deleted must also have all associated relationships explicitly deleted.</td>
</tr>
<tr>
<td>DETACH DELETE</td>
<td>Writing</td>
<td>Delete a node or set of nodes. All associated relationships will automatically be deleted.</td>
</tr>
<tr>
<td>DROP CONSTRAINT name</td>
<td>Schema</td>
<td>Drop a constraint using the name.</td>
</tr>
<tr>
<td>DROP INDEX name</td>
<td>Schema</td>
<td>Drop an index using the name.</td>
</tr>
<tr>
<td>FOREACH</td>
<td>Writing</td>
<td>Update data within a list, whether components of a path, or the result of aggregation.</td>
</tr>
<tr>
<td>LIMIT</td>
<td>Reading sub-clause</td>
<td>A sub-clause used to constrain the number of rows in the output.</td>
</tr>
<tr>
<td>LOAD CSV</td>
<td>Importing data</td>
<td>Use when importing data from CSV files.</td>
</tr>
<tr>
<td>MATCH</td>
<td>Reading</td>
<td>Specify the patterns to search for in the database.</td>
</tr>
<tr>
<td>MERGE</td>
<td>Reading/Writing</td>
<td>Ensures that a pattern exists in the graph. Either the pattern already exists, or it needs to be created.</td>
</tr>
<tr>
<td>ON CREATE</td>
<td>Reading/Writing</td>
<td>Used in conjunction with MERGE, specifying the actions to take if the pattern needs to be created.</td>
</tr>
<tr>
<td>ON MATCH</td>
<td>Reading/Writing</td>
<td>Used in conjunction with MERGE, specifying the actions to take if the pattern already exists.</td>
</tr>
<tr>
<td>OPTIONAL MATCH</td>
<td>Reading</td>
<td>Specify the patterns to search for in the database while using nulls for missing parts of the pattern.</td>
</tr>
<tr>
<td>ORDER BY [ASCENDING]</td>
<td>Reading sub-clause</td>
<td>A sub-clause following RETURN or WITH, specifying that the output should be sorted in either ascending (the default) or descending order.</td>
</tr>
<tr>
<td>REMOVE</td>
<td>Writing</td>
<td>Remove properties and labels from nodes and relationships.</td>
</tr>
<tr>
<td>RETURN ... [AS]</td>
<td>Projecting</td>
<td>Defines what to include in the query result set.</td>
</tr>
<tr>
<td>SET</td>
<td>Writing</td>
<td>Update labels on nodes and properties on nodes and relationships.</td>
</tr>
<tr>
<td>Clause</td>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>SKIP</td>
<td>Reading/Writing</td>
<td>A sub-clause defining from which row to start including the rows in the output.</td>
</tr>
<tr>
<td>UNION</td>
<td>Set operations</td>
<td>Combines the result of multiple queries. Duplicates are removed.</td>
</tr>
<tr>
<td>UNION ALL</td>
<td>Set operations</td>
<td>Combines the result of multiple queries. Duplicates are retained.</td>
</tr>
<tr>
<td>UNWIND ... [AS]</td>
<td>Projecting</td>
<td>Expands a list into a sequence of rows.</td>
</tr>
<tr>
<td>USE</td>
<td>Multiple graphs</td>
<td>Determines which graph a query, or query part, is executed against.</td>
</tr>
<tr>
<td>USING INDEX variable:Label(property)</td>
<td>Hint</td>
<td>Index hints are used to specify which index, if any, the planner should use as a starting point.</td>
</tr>
<tr>
<td>USING INDEX SEEK variable:Label(property)</td>
<td>Hint</td>
<td>Index seek hint instructs the planner to use an index seek for this clause.</td>
</tr>
<tr>
<td>USING JOIN ON variable</td>
<td>Hint</td>
<td>Join hints are used to enforce a join operation at specified points.</td>
</tr>
<tr>
<td>USING PERIODIC COMMIT</td>
<td>Hint</td>
<td>This query hint may be used to prevent an out-of-memory error from occurring when importing large amounts of data using LOAD CSV.</td>
</tr>
<tr>
<td>USING SCAN variable:Label</td>
<td>Hint</td>
<td>Scan hints are used to force the planner to do a label scan (followed by a filtering operation) instead of using an index.</td>
</tr>
<tr>
<td>WITH ... [AS]</td>
<td>Projecting</td>
<td>Allows query parts to be chained together, piping the results from one to be used as starting points or criteria in the next.</td>
</tr>
<tr>
<td>WHERE</td>
<td>Reading sub-clause</td>
<td>A sub-clause used to add constraints to the patterns in a MATCH or OPTIONAL MATCH clause, or to filter the results of a WITH clause.</td>
</tr>
<tr>
<td>WHERE EXISTS (...)</td>
<td>Reading sub-clause</td>
<td>An existential sub-query used to filter the results of a MATCH, OPTIONAL MATCH or WITH clause.</td>
</tr>
</tbody>
</table>

### 16.2. Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>Mathematical</td>
<td>Modulo division</td>
</tr>
<tr>
<td>*</td>
<td>Mathematical</td>
<td>Multiplication</td>
</tr>
<tr>
<td>*</td>
<td>Temporal</td>
<td>Multiplying a duration with a number</td>
</tr>
<tr>
<td>Operator</td>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>+</td>
<td>Mathematical</td>
<td>Addition</td>
</tr>
<tr>
<td>+</td>
<td>String</td>
<td>Concatenation</td>
</tr>
<tr>
<td>&lt;&lt;query-operators-property, +→&gt;</td>
<td>Property</td>
<td>Property mutation</td>
</tr>
<tr>
<td>+</td>
<td>List</td>
<td>Concatenation</td>
</tr>
<tr>
<td>+</td>
<td>Temporal</td>
<td>Adding two durations, or a duration and a temporal instant</td>
</tr>
<tr>
<td>&lt;&lt;query-operators-mathematical, →&gt;</td>
<td>Mathematical</td>
<td>Subtraction or unary minus</td>
</tr>
<tr>
<td>&lt;&lt;query-operators-temporal, →&gt;</td>
<td>Temporal</td>
<td>Subtracting a duration from a temporal instant or from another duration</td>
</tr>
<tr>
<td>.</td>
<td>Map</td>
<td>Static value access by key</td>
</tr>
<tr>
<td>.</td>
<td>Property</td>
<td>Static property access</td>
</tr>
<tr>
<td>/</td>
<td>Mathematical</td>
<td>Division</td>
</tr>
<tr>
<td>/</td>
<td>Temporal</td>
<td>Dividing a duration by a number</td>
</tr>
<tr>
<td>&lt;</td>
<td>Comparison</td>
<td>Less than</td>
</tr>
<tr>
<td>&lt;&lt;query-operators-comparison, &lt;→&gt;</td>
<td>Comparison</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Comparison</td>
<td>Inequality</td>
</tr>
<tr>
<td>&lt;&lt;query-operators-comparison, →&gt;</td>
<td>Comparison</td>
<td>Equality</td>
</tr>
<tr>
<td>&lt;&lt;query-operators-property, →&gt;</td>
<td>Property</td>
<td>Property replacement</td>
</tr>
<tr>
<td>=~</td>
<td>String</td>
<td>Regular expression match</td>
</tr>
<tr>
<td>&gt;</td>
<td>Comparison</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;&lt;query-operators-comparison, &gt;→&gt;</td>
<td>Comparison</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>AND</td>
<td>Boolean</td>
<td>Conjunction</td>
</tr>
<tr>
<td>CONTAINS</td>
<td>String</td>
<td>Case-sensitive inclusion search</td>
</tr>
<tr>
<td>DISTINCT</td>
<td>Aggregation</td>
<td>Duplicate removal</td>
</tr>
<tr>
<td>ENDS WITH</td>
<td>String</td>
<td>Case-sensitive suffix search</td>
</tr>
<tr>
<td>IN</td>
<td>List</td>
<td>List element existence check</td>
</tr>
<tr>
<td>IS NOT NULL</td>
<td>Comparison</td>
<td>Non-null check</td>
</tr>
<tr>
<td>IS NULL</td>
<td>Comparison</td>
<td>null check</td>
</tr>
<tr>
<td>NOT</td>
<td>Boolean</td>
<td>Negation</td>
</tr>
<tr>
<td>OR</td>
<td>Boolean</td>
<td>Disjunction</td>
</tr>
<tr>
<td>STARTS WITH</td>
<td>String</td>
<td>Case-sensitive prefix search</td>
</tr>
<tr>
<td>XOR</td>
<td>Boolean</td>
<td>Exclusive disjunction</td>
</tr>
<tr>
<td>[]</td>
<td>Map</td>
<td>Subscript (dynamic value access by key)</td>
</tr>
<tr>
<td>Operator</td>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>[ ]</td>
<td>Property</td>
<td>Subscript (dynamic property access)</td>
</tr>
<tr>
<td>[ ]</td>
<td>List</td>
<td>Subscript (accessing element(s) in a list)</td>
</tr>
<tr>
<td>^</td>
<td>Mathematical</td>
<td>Exponentiation</td>
</tr>
</tbody>
</table>

### 16.3. Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs()</td>
<td>Numeric</td>
<td>Returns the absolute value of a number.</td>
</tr>
<tr>
<td>acos()</td>
<td>Trigonometric</td>
<td>Returns the arccosine of a number in radians.</td>
</tr>
<tr>
<td>all()</td>
<td>Predicate</td>
<td>Tests whether the predicate holds for all elements in a list.</td>
</tr>
<tr>
<td>any()</td>
<td>Predicate</td>
<td>Tests whether the predicate holds for at least one element in a list.</td>
</tr>
<tr>
<td>asin()</td>
<td>Trigonometric</td>
<td>Returns the arcsine of a number in radians.</td>
</tr>
<tr>
<td>atan()</td>
<td>Trigonometric</td>
<td>Returns the arctangent of a number in radians.</td>
</tr>
<tr>
<td>atan2()</td>
<td>Trigonometric</td>
<td>Returns the arctangent2 of a set of coordinates in radians.</td>
</tr>
<tr>
<td>avg()</td>
<td>Aggregating</td>
<td>Returns the average of a set of values.</td>
</tr>
<tr>
<td>ceil()</td>
<td>Numeric</td>
<td>Returns the smallest floating point number that is greater than or equal to a number and equal to a mathematical integer.</td>
</tr>
<tr>
<td>coalesce()</td>
<td>Scalar</td>
<td>Returns the first non-<code>null</code> value in a list of expressions.</td>
</tr>
<tr>
<td>collect()</td>
<td>Aggregating</td>
<td>Returns a list containing the values returned by an expression.</td>
</tr>
<tr>
<td>cos()</td>
<td>Trigonometric</td>
<td>Returns the cosine of a number.</td>
</tr>
<tr>
<td>cot()</td>
<td>Trigonometric</td>
<td>Returns the cotangent of a number.</td>
</tr>
<tr>
<td>count()</td>
<td>Aggregating</td>
<td>Returns the number of values or rows.</td>
</tr>
<tr>
<td>date()</td>
<td>Temporal</td>
<td>Returns the current Date.</td>
</tr>
<tr>
<td>date(year [, month, day])</td>
<td>Temporal</td>
<td>Returns a calendar (Year-Month-Day) Date.</td>
</tr>
<tr>
<td>date(year [, week, dayOfWeek])</td>
<td>Temporal</td>
<td>Returns a week (Year-Week-Day) Date.</td>
</tr>
<tr>
<td>date(year [, quarter, dayOfQuarter])</td>
<td>Temporal</td>
<td>Returns a quarter (Year-Quarter-Day) Date.</td>
</tr>
<tr>
<td>Function</td>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>date({year [, ordinalDay]})</td>
<td>Temporal</td>
<td>Returns an ordinal (Year-Day) Date.</td>
</tr>
<tr>
<td>date(string)</td>
<td>Temporal</td>
<td>Returns a Date by parsing a string.</td>
</tr>
<tr>
<td>date({map})</td>
<td>Temporal</td>
<td>Returns a Date from a map of another temporal value's components.</td>
</tr>
<tr>
<td>date.realtime()</td>
<td>Temporal</td>
<td>Returns the current Date using the <code>realtime</code> clock.</td>
</tr>
<tr>
<td>date.statement()</td>
<td>Temporal</td>
<td>Returns the current Date using the <code>statement</code> clock.</td>
</tr>
<tr>
<td>date.transaction()</td>
<td>Temporal</td>
<td>Returns the current Date using the <code>transaction</code> clock.</td>
</tr>
<tr>
<td>date.truncate()</td>
<td>Temporal</td>
<td>Returns a Date obtained by truncating a value at a specific component boundary. Truncation summary.</td>
</tr>
<tr>
<td>datetime()</td>
<td>Temporal</td>
<td>Returns the current DateTime.</td>
</tr>
<tr>
<td>datetime({year [, month, day, ...]})</td>
<td>Temporal</td>
<td>Returns a calendar (Year-Month-Day) DateTime.</td>
</tr>
<tr>
<td>datetime({year [, week, dayOfWeek, ...]})</td>
<td>Temporal</td>
<td>Returns a week (Year-Week-Day) DateTime.</td>
</tr>
<tr>
<td>datetime({year [, quarter, dayOfQuarter, ...]})</td>
<td>Temporal</td>
<td>Returns a quarter (Year-Quarter-Day) DateTime.</td>
</tr>
<tr>
<td>datetime({year [, ordinalDay, ...]})</td>
<td>Temporal</td>
<td>Returns an ordinal (Year-Day) DateTime.</td>
</tr>
<tr>
<td>datetime(string)</td>
<td>Temporal</td>
<td>Returns a DateTime by parsing a string.</td>
</tr>
<tr>
<td>datetime({map})</td>
<td>Temporal</td>
<td>Returns a DateTime from a map of another temporal value's components.</td>
</tr>
<tr>
<td>datetime({epochSeconds})</td>
<td>Temporal</td>
<td>Returns a DateTime from a timestamp.</td>
</tr>
<tr>
<td>datetime.realtime()</td>
<td>Temporal</td>
<td>Returns the current DateTime using the <code>realtime</code> clock.</td>
</tr>
<tr>
<td>datetime.statement()</td>
<td>Temporal</td>
<td>Returns the current DateTime using the <code>statement</code> clock.</td>
</tr>
<tr>
<td>datetime.transaction()</td>
<td>Temporal</td>
<td>Returns the current DateTime using the <code>transaction</code> clock.</td>
</tr>
<tr>
<td>datetime.truncate()</td>
<td>Temporal</td>
<td>Returns a DateTime obtained by truncating a value at a specific component boundary. Truncation summary.</td>
</tr>
<tr>
<td>degrees()</td>
<td>Trigonometric</td>
<td>Converts radians to degrees.</td>
</tr>
<tr>
<td>Function</td>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>distance()</td>
<td>Spatial</td>
<td>Returns a floating point number representing the geodesic distance between</td>
</tr>
<tr>
<td></td>
<td></td>
<td>any two points in the same CRS.</td>
</tr>
<tr>
<td>duration(map)</td>
<td>Temporal</td>
<td>Returns a Duration from a map of its components.</td>
</tr>
<tr>
<td>duration(string)</td>
<td>Temporal</td>
<td>Returns a Duration by parsing a string.</td>
</tr>
<tr>
<td>duration.between</td>
<td>Temporal</td>
<td>Returns a Duration equal to the difference between two given instants.</td>
</tr>
<tr>
<td>duration.inDays</td>
<td>Temporal</td>
<td>Returns a Duration equal to the difference in whole days or weeks between</td>
</tr>
<tr>
<td></td>
<td></td>
<td>two given instants.</td>
</tr>
<tr>
<td>duration.inMonths</td>
<td>Temporal</td>
<td>Returns a Duration equal to the difference in whole months, quarters or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>years between two given instants.</td>
</tr>
<tr>
<td>duration.inSeconds</td>
<td>Temporal</td>
<td>Returns a Duration equal to the difference in seconds and fractions of</td>
</tr>
<tr>
<td></td>
<td></td>
<td>seconds, or minutes or hours, between two given instants.</td>
</tr>
<tr>
<td>e()</td>
<td>Logarithmic</td>
<td>Returns the base of the natural logarithm, (e).</td>
</tr>
<tr>
<td>endNode()</td>
<td>Scalar</td>
<td>Returns the end node of a relationship.</td>
</tr>
<tr>
<td>exists()</td>
<td>Predicate</td>
<td>Returns true if a match for the pattern exists in the graph, or if the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>specified property exists in the node, relationship or map.</td>
</tr>
<tr>
<td>exp()</td>
<td>Logarithmic</td>
<td>Returns (e^n), where (e) is the base of the natural logarithm, and (n) is the value of the argument expression.</td>
</tr>
<tr>
<td>floor()</td>
<td>Numeric</td>
<td>Returns the largest floating point number that is less than or equal to a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>number and equal to a mathematical integer.</td>
</tr>
<tr>
<td>haversin()</td>
<td>Trigonometric</td>
<td>Returns half the versine of a number.</td>
</tr>
<tr>
<td>head()</td>
<td>Scalar</td>
<td>Returns the first element in a list.</td>
</tr>
<tr>
<td>id()</td>
<td>Scalar</td>
<td>Returns the id of a relationship or node.</td>
</tr>
<tr>
<td>keys()</td>
<td>List</td>
<td>Returns a list containing the string representations for all the property</td>
</tr>
<tr>
<td></td>
<td></td>
<td>names of a node, relationship, or map.</td>
</tr>
<tr>
<td>labels()</td>
<td>List</td>
<td>Returns a list containing the string representations for all the labels of</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a node.</td>
</tr>
<tr>
<td>Function</td>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>last()</td>
<td>Scalar</td>
<td>Returns the last element in a list.</td>
</tr>
<tr>
<td>left()</td>
<td>String</td>
<td>Returns a string containing the specified number of leftmost characters of the original string.</td>
</tr>
<tr>
<td>length()</td>
<td>Scalar</td>
<td>Returns the length of a path.</td>
</tr>
<tr>
<td>localdatetime()</td>
<td>Temporal</td>
<td>Returns the current LocalDateTime.</td>
</tr>
<tr>
<td>localdatetime([year [, month, day, ...]])</td>
<td>Temporal</td>
<td>Returns a calendar (Year-Month-Day) LocalDateTime.</td>
</tr>
<tr>
<td>localdatetime([year [, week, dayOfWeek, ...]])</td>
<td>Temporal</td>
<td>Returns a week (Year-Week-Day) LocalDateTime.</td>
</tr>
<tr>
<td>localdatetime([year [, quarter, dayOfQuarter, ...]])</td>
<td>Temporal</td>
<td>Returns a quarter (Year-Quarter-Day) DateTime.</td>
</tr>
<tr>
<td>localdatetime([year [, ordinalDay, ...]])</td>
<td>Temporal</td>
<td>Returns an ordinal (Year-Day) LocalDateTime.</td>
</tr>
<tr>
<td>localdatetime(string)</td>
<td>Temporal</td>
<td>Returns a LocalDateTime by parsing a string.</td>
</tr>
<tr>
<td>localdatetime([map])</td>
<td>Temporal</td>
<td>Returns a LocalDateTime from a map of another temporal value's components.</td>
</tr>
<tr>
<td>localdatetime.realtime()</td>
<td>Temporal</td>
<td>Returns the current LocalDateTime using the realtime clock.</td>
</tr>
<tr>
<td>localdatetime.statement()</td>
<td>Temporal</td>
<td>Returns the current LocalDateTime using the statement clock.</td>
</tr>
<tr>
<td>localdatetime.transaction()</td>
<td>Temporal</td>
<td>Returns the current LocalDateTime using the transaction clock.</td>
</tr>
<tr>
<td>localdatetime.truncate()</td>
<td>Temporal</td>
<td>Returns a LocalDateTime obtained by truncating a value at a specific component boundary. Truncation summary.</td>
</tr>
<tr>
<td>localtime()</td>
<td>Temporal</td>
<td>Returns the current LocalTime.</td>
</tr>
<tr>
<td>localtime([hour [, minute, second, ...]])</td>
<td>Temporal</td>
<td>Returns a LocalTime with the specified component values.</td>
</tr>
<tr>
<td>localtime(string)</td>
<td>Temporal</td>
<td>Returns a LocalTime by parsing a string.</td>
</tr>
<tr>
<td>localtime([time [, hour, ...]])</td>
<td>Temporal</td>
<td>Returns a LocalTime from a map of another temporal value's components.</td>
</tr>
<tr>
<td>localtime.realtime()</td>
<td>Temporal</td>
<td>Returns the current LocalTime using the realtime clock.</td>
</tr>
<tr>
<td>localtime.statement()</td>
<td>Temporal</td>
<td>Returns the current LocalTime using the statement clock.</td>
</tr>
<tr>
<td>localtime.transaction()</td>
<td>Temporal</td>
<td>Returns the current LocalTime using the transaction clock.</td>
</tr>
<tr>
<td>Function</td>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>----------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>localtime.truncate</td>
<td>Temporal</td>
<td>Returns a LocalTime obtained by truncating a value at a specific component boundary. <a href="#">Truncation summary</a>.</td>
</tr>
<tr>
<td>log</td>
<td>Logarithmic</td>
<td>Returns the natural logarithm of a number.</td>
</tr>
<tr>
<td>log10</td>
<td>Logarithmic</td>
<td>Returns the common logarithm (base 10) of a number.</td>
</tr>
<tr>
<td>lTrim</td>
<td>String</td>
<td>Returns the original string with leading whitespace removed.</td>
</tr>
<tr>
<td>max</td>
<td>Aggregating</td>
<td>Returns the maximum value in a set of values.</td>
</tr>
<tr>
<td>min</td>
<td>Aggregating</td>
<td>Returns the minimum value in a set of values.</td>
</tr>
<tr>
<td>nodes</td>
<td>List</td>
<td>Returns a list containing all the nodes in a path.</td>
</tr>
<tr>
<td>none</td>
<td>Predicate</td>
<td>Returns true if the predicate holds for no element in a list.</td>
</tr>
<tr>
<td>percentileCont</td>
<td>Aggregating</td>
<td>Returns the percentile of the given value over a group using linear interpolation.</td>
</tr>
<tr>
<td>percentileDisc</td>
<td>Aggregating</td>
<td>Returns the nearest value to the given percentile over a group using a rounding method.</td>
</tr>
<tr>
<td>pi</td>
<td>Trigonometric</td>
<td>Returns the mathematical constant ( \pi ).</td>
</tr>
<tr>
<td>point() - Cartesian 2D</td>
<td>Spatial</td>
<td>Returns a 2D point object, given two coordinate values in the Cartesian coordinate system.</td>
</tr>
<tr>
<td>point() - Cartesian 3D</td>
<td>Spatial</td>
<td>Returns a 3D point object, given three coordinate values in the Cartesian coordinate system.</td>
</tr>
<tr>
<td>point() - WGS 84 2D</td>
<td>Spatial</td>
<td>Returns a 2D point object, given two coordinate values in the WGS 84 coordinate system.</td>
</tr>
<tr>
<td>point() - WGS 84 3D</td>
<td>Spatial</td>
<td>Returns a 3D point object, given three coordinate values in the WGS 84 coordinate system.</td>
</tr>
<tr>
<td>properties</td>
<td>Scalar</td>
<td>Returns a map containing all the properties of a node or relationship.</td>
</tr>
<tr>
<td>radians</td>
<td>Trigonometric</td>
<td>Converts degrees to radians.</td>
</tr>
<tr>
<td>rand</td>
<td>Numeric</td>
<td>Returns a random floating point number in the range from 0 (inclusive) to 1 (exclusive); i.e. ([0, 1)).</td>
</tr>
<tr>
<td>Function</td>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>randomUUID()</td>
<td>Scalar</td>
<td>Returns a string value corresponding to a randomly-generated UUID.</td>
</tr>
<tr>
<td>range()</td>
<td>List</td>
<td>Returns a list comprising all integer values within a specified range.</td>
</tr>
<tr>
<td>reduce()</td>
<td>List</td>
<td>Runs an expression against individual elements of a list, storing the result of the expression in an accumulator.</td>
</tr>
<tr>
<td>relationships()</td>
<td>List</td>
<td>Returns a list containing all the relationships in a path.</td>
</tr>
<tr>
<td>replace()</td>
<td>String</td>
<td>Returns a string in which all occurrences of a specified string in the original string have been replaced by another (specified) string.</td>
</tr>
<tr>
<td>reverse()</td>
<td>List</td>
<td>Returns a list in which the order of all elements in the original list have been reversed.</td>
</tr>
<tr>
<td>reverse()</td>
<td>String</td>
<td>Returns a string in which the order of all characters in the original string have been reversed.</td>
</tr>
<tr>
<td>right()</td>
<td>String</td>
<td>Returns a string containing the specified number of rightmost characters of the original string.</td>
</tr>
<tr>
<td>round()</td>
<td>Numeric</td>
<td>Returns the value of a number rounded to the nearest integer.</td>
</tr>
<tr>
<td>rTrim()</td>
<td>String</td>
<td>Returns the original string with trailing whitespace removed.</td>
</tr>
<tr>
<td>sign()</td>
<td>Numeric</td>
<td>Returns the signum of a number: 0 if the number is 0, -1 for any negative number, and 1 for any positive number.</td>
</tr>
<tr>
<td>sin()</td>
<td>Trigonometric</td>
<td>Returns the sine of a number.</td>
</tr>
<tr>
<td>single()</td>
<td>Predicate</td>
<td>Returns true if the predicate holds for exactly one of the elements in a list.</td>
</tr>
<tr>
<td>size()</td>
<td>Scalar</td>
<td>Returns the number of items in a list.</td>
</tr>
<tr>
<td>size() applied to pattern expression</td>
<td>Scalar</td>
<td>Returns the number of paths matching the pattern expression.</td>
</tr>
<tr>
<td>size() applied to string</td>
<td>Scalar</td>
<td>Returns the number of Unicode characters in a string.</td>
</tr>
<tr>
<td>split()</td>
<td>String</td>
<td>Returns a list of strings resulting from the splitting of the original string around matches of the given delimiter.</td>
</tr>
<tr>
<td>sqrt()</td>
<td>Logarithmic</td>
<td>Returns the square root of a number.</td>
</tr>
<tr>
<td>Function</td>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>startNode()</td>
<td>Scalar</td>
<td>Returns the start node of a relationship.</td>
</tr>
<tr>
<td>stDev()</td>
<td>Aggregating</td>
<td>Returns the standard deviation for the given value over a group for a sample of a population.</td>
</tr>
<tr>
<td>stDevP()</td>
<td>Aggregating</td>
<td>Returns the standard deviation for the given value over a group for an entire population.</td>
</tr>
<tr>
<td>substring()</td>
<td>String</td>
<td>Returns a substring of the original string, beginning with a 0-based index start and length.</td>
</tr>
<tr>
<td>sum()</td>
<td>Aggregating</td>
<td>Returns the sum of a set of numeric values.</td>
</tr>
<tr>
<td>tail()</td>
<td>List</td>
<td>Returns all but the first element in a list.</td>
</tr>
<tr>
<td>tan()</td>
<td>Trigonometric</td>
<td>Returns the tangent of a number.</td>
</tr>
<tr>
<td>time()</td>
<td>Temporal</td>
<td>Returns the current Time.</td>
</tr>
<tr>
<td>time({hour [, minute, ...]})</td>
<td>Temporal</td>
<td>Returns a Time with the specified component values.</td>
</tr>
<tr>
<td>time(string)</td>
<td>Temporal</td>
<td>Returns a Time by parsing a string.</td>
</tr>
<tr>
<td>time({time [, hour, ..., timezone]})</td>
<td>Temporal</td>
<td>Returns a Time from a map of another temporal value's components.</td>
</tr>
<tr>
<td>time.realtime()</td>
<td>Temporal</td>
<td>Returns the current Time using the realtime clock.</td>
</tr>
<tr>
<td>time.statement()</td>
<td>Temporal</td>
<td>Returns the current Time using the statement clock.</td>
</tr>
<tr>
<td>time.transaction()</td>
<td>Temporal</td>
<td>Returns the current Time using the transaction clock.</td>
</tr>
<tr>
<td>time.truncate()</td>
<td>Temporal</td>
<td>Returns a Time obtained by truncating a value at a specific component boundary. Truncation summary.</td>
</tr>
<tr>
<td>timestamp()</td>
<td>Scalar</td>
<td>Returns the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC.</td>
</tr>
<tr>
<td>toBoolean()</td>
<td>Scalar</td>
<td>Converts a string value to a boolean value.</td>
</tr>
<tr>
<td>toFloat()</td>
<td>Scalar</td>
<td>Converts an integer or string value to a floating point number.</td>
</tr>
<tr>
<td>toInteger()</td>
<td>Scalar</td>
<td>Converts a floating point or string value to an integer value.</td>
</tr>
<tr>
<td>toLower()</td>
<td>String</td>
<td>Returns the original string in lowercase.</td>
</tr>
<tr>
<td>Function</td>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>toString()</td>
<td>String</td>
<td>Converts an integer, float, boolean or temporal (i.e. Date, Time, LocalTime, DateTime, LocalDate or Duration) value to a string.</td>
</tr>
<tr>
<td>toUpper()</td>
<td>String</td>
<td>Returns the original string in uppercase.</td>
</tr>
<tr>
<td>trim()</td>
<td>String</td>
<td>Returns the original string with leading and trailing whitespace removed.</td>
</tr>
<tr>
<td>type()</td>
<td>Scalar</td>
<td>Returns the string representation of the relationship type.</td>
</tr>
</tbody>
</table>

16.4. Expressions

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASE Expression</td>
<td>A generic conditional expression, similar to if/else statements available in other languages.</td>
</tr>
</tbody>
</table>

16.5. Cypher query options

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CYPHER $version query</td>
<td>Version</td>
<td>This will force 'query' to use Neo4j Cypher $version. The default is 4.0.</td>
</tr>
<tr>
<td>CYPHER runtime=interpreted query</td>
<td>Runtime</td>
<td>This will force the query planner to use the interpreted runtime. This is the only option in Neo4j Community Edition.</td>
</tr>
<tr>
<td>CYPHER runtime=slotted query</td>
<td>Runtime</td>
<td>This will cause the query planner to use the slotted runtime. This is only available in Neo4j Enterprise Edition.</td>
</tr>
<tr>
<td>CYPHER runtime=pipelined query</td>
<td>Runtime</td>
<td>This will cause the query planner to use the pipelined runtime if it supports 'query'. This is only available in Neo4j Enterprise Edition.</td>
</tr>
</tbody>
</table>

16.6. Administrative commands

The following commands are only executable against the system database:

<table>
<thead>
<tr>
<th>Command</th>
<th>Admin category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTER CURRENT USER SET PASSWORD FROM ... TO</td>
<td>User and role</td>
<td>Change the password of the user that is currently logged in.</td>
</tr>
<tr>
<td>Command</td>
<td>Admin category</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>ALTER USER ... [SET PASSWORD [password [CHANGE [NOT] REQUIRED]</td>
<td>CHANGE [NOT] REQUIRED]] [SET STATUS {ACTIVE</td>
<td>SUSPENDED}]]</td>
</tr>
<tr>
<td>CREATE [OR REPLACE] DATABASE ... [IF NOT EXISTS]]</td>
<td>Database</td>
<td>Creates a new database.</td>
</tr>
<tr>
<td>CREATE [OR REPLACE] ROLE ... [IF NOT EXISTS] [AS COPY OF]]</td>
<td>User and role</td>
<td>Creates new roles.</td>
</tr>
<tr>
<td>CREATE [OR REPLACE] USER ... [IF NOT EXISTS]</td>
<td>User and role</td>
<td>Creates a new user and sets the password for the new account. Optionally the account status can also be set and if the user should change the password upon first login.</td>
</tr>
<tr>
<td>DENY ... ON DATABASE ... TO</td>
<td>Privilege</td>
<td>Denies a database or schema privilege to one or multiple roles.</td>
</tr>
<tr>
<td>DENY ... ON DBMS TO</td>
<td>Privilege</td>
<td>Denies a DBMS privilege to one or multiple roles.</td>
</tr>
<tr>
<td>DENY ... ON GRAPH ... [NODES</td>
<td>RELATIONSHIPS</td>
<td>ELEMENTS] ... TO</td>
</tr>
<tr>
<td>DROP DATABASE ... [IF EXISTS]]</td>
<td>Database</td>
<td>Deletes a specified database.</td>
</tr>
<tr>
<td>DROP ROLE ... [IF EXISTS]</td>
<td>User and role</td>
<td>Deletes a specified role.</td>
</tr>
<tr>
<td>DROP USER ... [IF EXISTS]]</td>
<td>User and role</td>
<td>Deletes a specified user.</td>
</tr>
<tr>
<td>GRANT ... ON DATABASE ... TO</td>
<td>Privilege</td>
<td>Assigns a database or schema privilege to one or multiple roles.</td>
</tr>
<tr>
<td>GRANT ... ON DBMS TO</td>
<td>Privilege</td>
<td>Assigns a DBMS privilege to one or multiple roles.</td>
</tr>
<tr>
<td>GRANT ... ON GRAPH ... [NODES</td>
<td>RELATIONSHIPS</td>
<td>ELEMENTS] ... TO</td>
</tr>
<tr>
<td>GRANT ROLE[S]] ... TO</td>
<td>User and role</td>
<td>Assigns one or multiple roles to one or multiple users.</td>
</tr>
<tr>
<td>SHOW [ROLE ...</td>
<td>USER ...</td>
<td>ALL \ PRIVILEGES]</td>
</tr>
<tr>
<td>START DATABASE</td>
<td>Database</td>
<td>Starts up a specified database.</td>
</tr>
<tr>
<td>STOP DATABASE</td>
<td>Database</td>
<td>Stops a specified database.</td>
</tr>
<tr>
<td>REVOKE [GRANT</td>
<td>DENY] ... ON DATABASE ... FROM</td>
<td>Privilege</td>
</tr>
<tr>
<td>REVOKE [GRANT</td>
<td>DENY] ... ON DBMS FROM]</td>
<td>Privilege</td>
</tr>
<tr>
<td>Command</td>
<td>Admin category</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>REVOKE [GRANT</td>
<td>DENY] ... ON GRAPH ... [NODES</td>
<td>RELATIONSHIPS</td>
</tr>
<tr>
<td>REVOKE ROLE[S] ... FROM</td>
<td>User and role</td>
<td>Removes one or multiple roles from one or multiple users.</td>
</tr>
<tr>
<td>SHOW [ALL</td>
<td>POPULATED] ROLES [WITH USERS]]</td>
<td>User and role</td>
</tr>
<tr>
<td>SHOW DATABASE</td>
<td>Database</td>
<td>Returns information about a specified database.</td>
</tr>
<tr>
<td>SHOW DATABASES</td>
<td>Database</td>
<td>Returns information about all databases.</td>
</tr>
<tr>
<td>SHOW DEFAULT DATABASE</td>
<td>Database</td>
<td>Returns information about the default database.</td>
</tr>
<tr>
<td>SHOW USERS</td>
<td>User and role</td>
<td>Returns information about all users.</td>
</tr>
</tbody>
</table>

16.7. Privilege Actions

<table>
<thead>
<tr>
<th>Name</th>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS</td>
<td>Database</td>
<td>Determines whether a user can access a specific database.</td>
</tr>
<tr>
<td>ALL DATABASE PRIVILEGES</td>
<td>Database and schema</td>
<td>Implies all privileges from the categories Database and Schema for a specific database.</td>
</tr>
<tr>
<td>ASSIGN ROLE</td>
<td>DBMS</td>
<td>Determines whether the user can grant roles.</td>
</tr>
<tr>
<td>CONSTRAINT MANAGEMENT</td>
<td>Schema</td>
<td>Determines whether a user is allowed to create and drop constraints on a specific database.</td>
</tr>
<tr>
<td>CREATE CONSTRAINT</td>
<td>Schema</td>
<td>Determines whether a user is allowed to create constraints on a specific database.</td>
</tr>
<tr>
<td>CREATE INDEX</td>
<td>Schema</td>
<td>Determines whether a user is allowed to create indexes on a specific database.</td>
</tr>
<tr>
<td>CREATE NEW NODE LABEL</td>
<td>Schema</td>
<td>Determines whether a user is allowed to create new node labels on a specific database.</td>
</tr>
<tr>
<td>CREATE NEW PROPERTY NAME</td>
<td>Schema</td>
<td>Determines whether a user is allowed to create new property names on a specific database.</td>
</tr>
<tr>
<td>CREATE NEW RELATIONSHIP TYPE</td>
<td>Schema</td>
<td>Determines whether a user is allowed to create new relationship types on a specific database.</td>
</tr>
<tr>
<td>Name</td>
<td>Category</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CREATE ROLE</td>
<td>DBMS</td>
<td>Determines whether the user can create new roles.</td>
</tr>
<tr>
<td>DROP CONSTRAINT</td>
<td>Schema</td>
<td>Determines whether a user is allowed to drop constraints on a specific database.</td>
</tr>
<tr>
<td>DROP INDEX</td>
<td>Schema</td>
<td>Determines whether a user is allowed to drop indexes on a specific database.</td>
</tr>
<tr>
<td>DROP ROLE</td>
<td>DBMS</td>
<td>Determines whether the user can delete roles.</td>
</tr>
<tr>
<td>INDEX MANAGEMENT</td>
<td>Schema</td>
<td>Determines whether a user is allowed to create and drop indexes on a specific database.</td>
</tr>
<tr>
<td>MATCH</td>
<td>GRAPH</td>
<td>Determines whether the properties of an element (node, relationship or both) can be read and the element can be found and traversed while executing queries on the specified graph.</td>
</tr>
<tr>
<td>NAME MANAGEMENT</td>
<td>Schema</td>
<td>Determines whether a user is allowed to create new labels, types and property names on a specific database.</td>
</tr>
<tr>
<td>READ</td>
<td>GRAPH</td>
<td>Determines whether the properties of an element (node, relationship or both) can be read while executing queries on the specified graph.</td>
</tr>
<tr>
<td>REMOVE ROLE</td>
<td>DBMS</td>
<td>Determines whether the user can revoke roles.</td>
</tr>
<tr>
<td>ROLE MANAGEMENT</td>
<td>DBMS</td>
<td>Determines whether the user can create, drop, grant, revoke and show roles.</td>
</tr>
<tr>
<td>SHOW ROLE</td>
<td>DBMS</td>
<td>Determines whether the user can get information about existing and assigned roles.</td>
</tr>
<tr>
<td>START</td>
<td>Database</td>
<td>Determines whether a user can start up a specific database.</td>
</tr>
<tr>
<td>STOP</td>
<td>Database</td>
<td>Determines whether a user can stop a specific running database.</td>
</tr>
<tr>
<td>TRAVERSE</td>
<td>GRAPH</td>
<td>Determines whether an element (node, relationship or both) can be found and traversed while executing queries on the specified graph.</td>
</tr>
<tr>
<td>WRITE</td>
<td>GRAPH</td>
<td>Determines whether the user can execute write operations on the specified graph.</td>
</tr>
</tbody>
</table>
Appendix A: Cypher styleguide

This appendix contains the recommended style when writing Cypher queries.

This appendix contains the following:

- General recommendations
- Indentations and line breaks
- Casing
- Spacing
- Patterns
- Meta characters

The purpose of the styleguide is to make the code as easy to read as possible, and thereby contributing to lower cost of maintenance.

For rules and recommendations for naming of labels, relationship types and properties, please see the Naming rules and recommendations.

A.1. General recommendations

- When using Cypher language constructs in prose, use a monospaced font and follow the styling rules.
- When referring to labels and relationship types, the colon should be included as follows: :Label, :REL_TYPE.
- When referring to functions, use lower camel case and parentheses should be used as follows: shortestPath(). Arguments should normally not be included.
- If you are storing Cypher statements in a separate file, use the file extension .cypher.

A.2. Indentation and line breaks

- Start a new clause on a new line.

  **Bad**

  ```cypher
  MATCH (n) WHERE n.name CONTAINS 's' RETURN n.name
  ```

  **Good**

  ```cypher
  MATCH (n)
  WHERE n.name CONTAINS 's'
  RETURN n.name
  ```

- Indent `ON CREATE` and `ON MATCH` with two spaces. Put `ON CREATE` before `ON MATCH` if both are present.
Bad

MERGE (n) ON CREATE SET n.prop = 0
MERGE (a:A)-[:T]->(b:B)
  ON MATCH SET b.name = 'you'
  ON CREATE SET a.name = 'me'
RETURN a.prop

Good

MERGE (n)
  ON CREATE SET n.prop = 0
MERGE (a:A)-[:T]->(b:B)
  ON CREATE SET a.name = 'me'
  ON MATCH SET b.name = 'you'
RETURN a.prop

- Start a subquery on a new line after the opening brace, indented with two (additional) spaces. Leave the closing brace on its own line.

Bad

MATCH (a:A)
WHERE
  EXISTS ( MATCH (a)->(b:B) WHERE b.prop = $param )
RETURN a.foo

Also bad

MATCH (a:A)
WHERE EXISTS
  (MATCH (a)->(b:B)
  WHERE b.prop = $param)
RETURN a.foo

Good

MATCH (a:A)
WHERE EXISTS {
  MATCH (a)->(b:B)
  WHERE b.prop = $param
}
RETURN a.foo

- Do not break the line if the simplified subquery form is used.

Bad

MATCH (a:A)
WHERE EXISTS {
  (a)->(b:B)
}
RETURN a.prop

Good

MATCH (a:A)
WHERE EXISTS ( (a)->(b:B) )
RETURN a.prop
A.3. Casing

• Write keywords in upper case.

**Bad**

```
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

```sql
CREATE (n {prop: 0})
WITH rand() AS rand, $param AS map
RETURN rand, map.propertyKey, count(n)
```

A.4. Spacing

- For literal maps:
  - No space between the opening brace and the first key
  - No space between key and colon
  - One space between colon and value
  - No space between value and comma
  - One space between comma and next key
  - No space between the last value and the closing brace

**Bad**

```sql
WITH { key1: 'value', key2: 42 } AS map
RETURN map
```

**Good**

```sql
WITH (key1: 'value', key2: 42) AS map
RETURN map
```

- One space between label/type predicates and property predicates in patterns.

**Bad**

```sql
MATCH (p:Person{property: -1})-[[:KNOWS] {since: 2016}]->()
RETURN p.name
```

**Good**

```sql
MATCH (p:Person {property: -1})-[[:KNOWS] {since: 2016}]->()
RETURN p.name
```

- No space in patterns.

**Bad**

```sql
MATCH (:Person) --> (:Vehicle)
RETURN count(*)
```

**Good**

```sql
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.
A.5. Patterns

- When patterns wrap lines, break after arrows, not before.

**Bad**

```
MATCH (:Person)-->(vehicle:Car)-->(:Company)
  <--(:Country)
RETURN count(vehicle)
```

**Good**

```
MATCH (:Person)-->(vehicle:Car)-->(:Company)<--
  (:Country)
RETURN count(vehicle)
```

- Use anonymous nodes and relationships when the variable would not be used.

**Bad**

```
CREATE (a:End {prop: 42}),
  (b:End {prop: 3}),
  (c:Begin {prop: id(a)})
```

**Good**

```
CREATE (a:End {prop: 42}),
  (:End {prop: 3}),
  (:Begin {prop: id(a)})
```

- Chain patterns together to avoid repeating variables.

**Bad**

```
MATCH (:Person)-->(vehicle:Car), (vehicle:Car)-->(:Company)
RETURN count(vehicle)
```

**Good**

```
MATCH (:Person)-->(vehicle:Car)-->(:Company)
RETURN count(vehicle)
```

- Put named nodes before anonymous nodes.
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

A.6. 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.
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.