

Legend

Syntax

MATCH ☑

MATCH (n:Person)-[:KNOWS]->(m:Person)

WHERE n.name = 'Alice'

Node patterns can contain labels and properties.

MATCH (n)-->(m)

Any pattern can be used in MATCH.

MATCH (n {name: 'Alice'})-->(m)

Patterns with node properties.

MATCH p = (n) --> (m)

Assign a path to p.

OPTIONAL MATCH (n)-[r]->(m)

Optional pattern: nulls will be used for missing parts.

Neo4j Cypher Refcard 4.2

Read
Write
General
Functions
Schema
Performance
Multidatabase
Security

Read query structure

[MATCH WHERE]
[OPTIONAL MATCH WHERE]
[WITH [ORDER BY] [SKIP] [LIMIT]]
RETURN [ORDER BY] [SKIP] [LIMIT]

WHERE C

WHERE n.property <> \$value

Use a predicate to filter. Note that WHERE is always part of a MATCH, OPTIONAL MATCH or WITH clause. Putting it after a different clause in a query will alter what it does.

WHERE EXISTS {
 MATCH (n)-->(m) WHERE n.age = m.age
}

Use an existential subquery to filter.

RETURN 🗗

RETURN *

Return the value of all variables.

RETURN n AS columnName

Use alias for result column name.

RETURN DISTINCT n

Return unique rows.

ORDER BY n.property

Sort the result.

ORDER BY n.property DESC

Sort the result in descending order.

SKIP \$skipNumber

Skip a number of results.

LIMIT \$limitNumber

Limit the number of results.

SKIP \$skipNumber LIMIT \$limitNumber

Skip results at the top and limit the number of results.

RETURN count(*)

The number of matching rows. See Aggregating functions for more.

WITH C

MATCH (user)-[:FRIEND]-(friend)

WHERE user.name = \$name

WITH user, count(friend) AS friends

WHERE friends > 10

RETURN user

The WITH syntax is similar to RETURN. It separates query parts explicitly, allowing you to declare which variables to carry over to the next part.

MATCH (user)-[:FRIEND]-(friend)

WITH user, count(friend) AS friends

ORDER BY friends DESC

SKIP 1

LIMIT 3

RETURN user

ORDER BY, SKIP, and LIMIT can also be used with WITH.

Read-write query structure

[MATCH WHERE]

[OPTIONAL MATCH WHERE]

[WITH [ORDER BY] [SKIP] [LIMIT]]

(CREATE | MERGE)*

[SET|DELETE|REMOVE|FOREACH]*

[RETURN [ORDER BY] [SKIP] [LIMIT]]

SET 🗗

SET n.property1 = \$value1,

n.property2 = \$value2

Update or create a property.

SET n = \$map

Set all properties. This will remove any existing properties.

SET n += \$map

Add and update properties, while keeping existing ones.

SET n:Person

Adds a label Person to a node.

UNION 🗗

MATCH (a)-[:KNOWS]->(b)

RETURN b.name

UNION

MATCH (a)-[:LOVES]->(b)

RETURN b.name

Returns the distinct union of all query results. Result column types and names have to match.

MATCH (a)-[:KNOWS]->(b)

RETURN b.name

UNION ALL

MATCH (a)-[:LOVES]->(b)

RETURN b.name

Returns the union of all query results, including duplicated rows.

Write-only query structure

(CREATE | MERGE)*

[SET|DELETE|REMOVE|FOREACH]*

[RETURN [ORDER BY] [SKIP] [LIMIT]]

CREATE [₹

CREATE (n {name: \$value})

Create a node with the given properties.

CREATE (n \$map)

Create a node with the given properties.

UNWIND \$listOfMaps AS properties

CREATE (n) SET n = properties

Create nodes with the given properties.

CREATE (n)-[r:KNOWS]->(m)

Create a relationship with the given type and direction; bind a variable to it.

CREATE (n)-[:LOVES {since: \$value}]->(m)

Create a relationship with the given type, direction, and properties.

MERGE ☑

MERGE (n:Person {name: \$value})

ON CREATE SET n.created = timestamp()

ON MATCH SET

n.counter = coalesce(n.counter, 0) + 1,

n.accessTime = timestamp()

Match a pattern or create it if it does not exist. Use ON CREATE and ON MATCH for conditional updates.

MATCH (a:Person {name: \$value1}),

(b:Person {name: \$value2})

MERGE (a)-[r:LOVES]->(b)

MERGE finds or creates a relationship between the nodes.

MATCH (a:Person {name: \$value1})

MERGE

(a)-[r:KNOWS]->(b:Person {name: \$value3})

MERGE finds or creates paths attached to the node.

DELETE [7

DELETE n, r

Delete a node and a relationship.

DETACH DELETE n

Delete a node and all relationships connected to it.

MATCH (n)

DETACH DELETE n

Delete all nodes and relationships from the database.

CALL subquery 🗗

```
CALL {
   MATCH (p:Person)-[:FRIEND_OF]->(other:Person)
RETURN p, other
   UNION
   MATCH (p:Child)-[:CHILD_OF]->(other:Parent)
RETURN p, other
}
```

This calls a subquery with two union parts. The result of the subquery can afterwards be post-processed.

REMOVE []

REMOVE n:Person

Remove a label from n.

REMOVE n.property

Remove a property.

FOREACH [₹

FOREACH (r IN relationships(path) |

SET r.marked = true)

Execute a mutating operation for each relationship in a path.

FOREACH (value IN coll |

CREATE (:Person {name: value}))

Execute a mutating operation for each element in a list.

CALL procedure 🗗

CALL db.labels() YIELD label

This shows a standalone call to the built-in procedure db.labels to list all labels used in the database. Note that required procedure arguments are given explicitly in brackets after the procedure name.

CALL java.stored.procedureWithArgs

Standalone calls may omit YIELD and also provide arguments implicitly via statement parameters, e.g. a standalone call requiring one argument input may be run by passing the parameter map {input: 'foo'}.

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

Calls the built-in procedure db.labels inside a larger query to count all labels used in the database. Calls inside a larger query always requires passing arguments and naming results explicitly with YIELD.

Import ☑	
LOAD CSV FROM	
'https://neo4j.com/docs/cypher-	
refcard/4.2/csv/artists.csv' AS line	
CREATE (:Artist {name: line[1], year:	
<pre>toInteger(line[2])})</pre>	
Load data from a CSV file and create nodes.	
LOAD CSV WITH HEADERS FROM	

LOAD CSV WITH HEADERS FROM		
https://neo4j.com/docs/cypher-		
refcard/4.2/csv/artists-with-headers.csv' AS		
line		
CREATE (:Artist {name: line.Name, year:		
<pre>toInteger(line.Year)})</pre>		

Load CSV data which has headers.

USING PERIODIC COMMIT 500

LOAD CSV WITH HEADERS FROM

'https://neo4j.com/docs/cypherrefcard/4.2/csv/artists-with-headers.csv' AS
line

CREATE (:Artist {name: line.Name, year:
toInteger(line.Year)})

Commit the current transaction after every 500 rows when importing large amounts of data.

LOAD CSV FROM 'https://neo4j.com/docs/cypherrefcard/4.2/csv/artists-fieldterminator.csv' AS line FIELDTERMINATOR ';' CREATE (:Artist {name: line[1], year: toInteger(line[2])})

Use a different field terminator, not the default which is a comma (with no whitespace around it).

LOAD CSV FROM 'https://neo4j.com/docs/cypherrefcard/4.2/csv/artists.csv' AS line RETURN DISTINCT file()

Returns the absolute path of the file that LOAD CSV is processing, returns null if called outside of LOAD CSV context.

LOAD CSV FROM		
'https://neo4j.com/docs/cypher-		
refcard/4.2/csv/artists.csv' AS line		
RETURN linenumber()		

Returns the line number that LOAD CSV is currently processing, returns null if called outside of LOAD CSV context.

Operators ☑		
General	DISTINCT, ., []	
Mathematical	+, -, *, /, %, ^	
Comparison	=, <>, <, >, <=, >=, IS NULL, IS NOT NULL	
Boolean	AND, OR, XOR, NOT	
String	+	
List	+, IN, [x], [x y]	
Regular Expression	=~	
String matching	STARTS WITH, ENDS WITH, CONTAINS	

null 🕜

- null is used to represent missing/undefined values.
- 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. To check if an expression is null, use IS NULL.
- Arithmetic expressions, comparisons and function calls (except coalesce) will return null if any argument is null.
- An attempt to access a missing element in a list or a property that doesn't exist yields null.
- In OPTIONAL MATCH clauses, nulls will be used for missing parts of the pattern.

Patterns 🗗

(n:Person)

Node with Person label.

(n:Person:Swedish)

Node with both Person and Swedish labels.

(n:Person {name: \$value})

Node with the declared properties.

()-[r {name: \$value}]-()

Matches relationships with the declared properties.

(n) - -> (m)

Relationship from n to m.

(n)--(m)

Relationship in any direction between n and m.

(n:Person)-->(m)

Node n labeled Person with relationship to m.

(m)<-[:KNOWS]-(n)

Relationship of type KNOWS from n to m.

(n)-[:KNOWS|:LOVES]->(m)

Relationship of type KNOWS or of type LOVES from n to m.

(n) - [r] - > (m)

Bind the relationship to variable Γ .

(n)-[*1..5]->(m)

Variable length path of between 1 and 5 relationships from n to m.

(n)-[*]->(m)

Variable length path of any number of relationships from n to m. (See Performance section.)

(n)-[:KNOWS]->(m {property: \$value})

A relationship of type KNOWS from a node n to a node m with the declared property.

shortestPath((n1:Person)-[*..6]-(n2:Person))

Find a single shortest path.

allShortestPaths((n1:Person)-[*..6]->

(n2:Person))

Find all shortest paths.

size((n)-->()-->())

Count the paths matching the pattern.

Labels

CREATE (n:Person {name: \$value})

Create a node with label and property.

MERGE (n:Person {name: \$value})

Matches or creates unique node(s) with the label and

property.

SET n:Spouse:Parent:Employee

Add label(s) to a node.

MATCH (n:Person)

Matches nodes labeled Person.

MATCH (n:Person)

WHERE n.name = \$value

Matches nodes labeled Person with the given name.

WHERE (n:Person)

Checks the existence of the label on the node.

labels(n)

Labels of the node.

REMOVE n:Person

Remove the label from the node.

Lists [조

['a', 'b', 'c'] AS list

Literal lists are declared in square brackets.

size(\$list) AS len, \$list[0] AS value Lists can be passed in as parameters.

range(\$firstNum, \$lastNum, \$step) AS list
range() creates a list of numbers (step is optional), other
functions returning lists are: labels(), nodes(),
relationships().

MATCH p = (a)-[:KNOWS*]->()
RETURN relationships(p) AS r

The list of relationships comprising a variable length path can be returned using named paths and relationships().

RETURN matchedNode.list[0] AS value, size(matchedNode.list) AS len

Properties can be lists of strings, numbers or booleans.

list[\$idx] AS value,

list[\$startIdx..\$endIdx] AS slice

List elements can be accessed with idx subscripts in square brackets. Invalid indexes return null. Slices can be retrieved with intervals from start_idx to end_idx, each of which can be omitted or negative. Out of range elements are ignored.

UNWIND \$names AS name MATCH (n {name: name}) RETURN avg(n.age)

With UNWIND, any list can be transformed back into individual rows. The example matches all names from a list of names.

MATCH (a)

RETURN [(a)-->(b) WHERE b.name = 'Bob' | b.age]
Pattern comprehensions may be used to do a custom projection from a match directly into a list.

MATCH (person)

RETURN person { .name, .age}

Map projections may be easily constructed from nodes, relationships and other map values.

Maps ☑

{name: 'Alice', age: 38,

address: {city: 'London', residential: true}}

Literal maps are declared in curly braces much like property maps. Lists are supported.

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

RETURN p.person.name

Access the property of a nested map.

MERGE (p:Person {name: \$map.name})

ON CREATE SET p = \$map

Maps can be passed in as parameters and used either as a map or by accessing keys.

MATCH (matchedNode:Person)

RETURN matchedNode

Nodes and relationships are returned as maps of their data.

map.name, map.age, map.children[0]

Map entries can be accessed by their keys. Invalid keys result in an error.

Predicates [7

n.property <> \$value

Use comparison operators.

exists(n.property)

Use functions.

n.number >= 1 AND n.number <= 10

Use boolean operators to combine predicates.

1 <= n.number <= 10

Use chained operators to combine predicates.

n:Person

Check for node labels.

variable IS NULL

Check if something is null.

NOT exists(n.property) OR n.property = \$value

Either the property does not exist or the predicate is true.

n.property = \$value

Non-existing property returns null, which is not equal to anything.

n["property"] = \$value

Properties may also be accessed using a dynamically computed property name.

n.property STARTS WITH 'Tim' OR

n.property ENDS WITH 'n' OR

n.property CONTAINS 'goodie'

String matching.

n.property =~ 'Tim.*'

String regular expression matching.

(n)-[:KNOWS]->(m)

Ensure the pattern has at least one match.

NOT (n)-[:KNOWS]->(m)

Exclude matches to (n)-[:KNOWS]->(m) from the result.

n.property IN [\$value1, \$value2]

Check if an element exists in a list.

List predicates 🗗

all(x IN coll WHERE exists(x.property))

Returns true if the predicate is true for all elements in

any(x IN coll WHERE exists(x.property))

Returns true if the predicate is true for at least one element in the list.

none(x IN coll WHERE exists(x.property))

Returns true if the predicate is false for all elements in the list

single(x IN coll WHERE exists(x.property))

Returns true if the predicate is true for exactly one element in the list.

CASE ☑

CASE n.eves

WHEN 'blue' THEN 1

WHEN 'brown' THEN 2

ELSE 3

END

Return THEN value from the matching WHEN value. The ELSE value is optional, and substituted for null if missing.

CASE

WHEN n.eyes = 'blue' THEN 1

WHEN n.age < 40 THEN 2

ELSE 3

END

Return THEN value from the first WHEN predicate evaluating to true. Predicates are evaluated in order.

List expressions 🗷

size(\$list)

Number of elements in the list.

reverse(\$list)

Reverse the order of the elements in the list.

head(\$list), last(\$list), tail(\$list)

head() returns the first, last() the last element of the list. tail() returns all but the first element. All return null for an empty list.

[x IN list | x.prop]

A list of the value of the expression for each element in the original list.

[x IN list WHERE x.prop <> \$value]

A filtered list of the elements where the predicate is true.

[x IN list WHERE x.prop <> \$value | x.prop]

A list comprehension that filters a list and extracts the value of the expression for each element in that list.

reduce(s = "", x IN list | s + x.prop)

Evaluate expression for each element in the list, accumulate the results.

Functions [7

coalesce(n.property, \$defaultValue)

The first non-null expression.

timestamp()

Milliseconds since midnight, January 1, 1970 UTC.

id(nodeOrRelationship)

The internal id of the relationship or node.

toInteger(\$expr)

Converts the given input into an integer if possible; otherwise it returns null.

toFloat(\$expr)

Converts the given input into a floating point number if possible; otherwise it returns null.

toBoolean(\$expr)

Converts the given input into a boolean if possible; otherwise it returns null.

kevs(\$expr)

Returns a list of string representations for the property names of a node, relationship, or map.

properties(\$expr)

Returns a map containing all the properties of a node or relationship.

Path functions 🗗

length(path)

The number of relationships in the path.

nodes(path)

The nodes in the path as a list.

relationships(path)

The relationships in the path as a list.

[x IN nodes(path) | x.prop]

Extract properties from the nodes in a path.

Spatial functions 🗗

point({x: \$x, y: \$y})

Returns a point in a 2D cartesian coordinate system.

point({latitude: \$y, longitude: \$x})

Returns a point in a 2D geographic coordinate system, with coordinates specified in decimal degrees.

point({x: \$x, y: \$y, z: \$z})

Returns a point in a 3D cartesian coordinate system.

point({latitude: \$y, longitude: \$x, height: \$z})

Returns a point in a 3D geographic coordinate system, with latitude and longitude in decimal degrees, and height in meters.

distance(point({x: \$x1, y: \$y1}), point({x: \$x2, y: \$y2}))

Returns a floating point number representing the linear distance between two points. The returned units will be the same as those of the point coordinates, and it will work for both 2D and 3D cartesian points.

distance(point({latitude: \$y1, longitude:

\$x1}), point({latitude: \$y2, longitude: \$x2}))

Returns the geodesic distance between two points in meters. It can be used for 3D geographic points as well.

Temporal functions 🗗

date("2018-04-05")

Returns a date parsed from a string.

localtime("12:45:30.25")

Returns a time with no time zone.

time("12:45:30.25+01:00")

Returns a time in a specified time zone.

localdatetime("2018-04-05T12:34:00")

Returns a datetime with no time zone.

datetime("2018-04-05T12:34:00[Europe/Berlin]")

Returns a datetime in the specified time zone.

datetime({epochMillis: 3360000})

Transforms 3360000 as a UNIX Epoch time into a normal datetime.

date({year: \$year, month: \$month, day: \$day})

All of the temporal functions can also be called with a map of named components. This example returns a date from year, month and day components. Each function supports a different set of possible components.

datetime({date: \$date, time: \$time})

Temporal types can be created by combining other types. This example creates a datetime from a date and a time.

date({date: \$datetime, day: 5})

Temporal types can be created by selecting from more complex types, as well as overriding individual components. This example creates a date by selecting from a datetime, as well as overriding the day component.

WITH date("2018-04-05") AS d

RETURN d.year, d.month, d.day, d.week,

d.dayOfWeek

Accessors allow extracting components of temporal types.

Duration functions <a>□ <a>□

duration("P1Y2M10DT12H45M30.25S")

Returns a duration of 1 year, 2 months, 10 days, 12 hours, 45 minutes and 30.25 seconds.

duration.between(\$date1,\$date2)

Returns a duration between two temporal instances.

WITH duration("P1Y2M10DT12H45M") AS d RETURN d.years, d.months, d.days, d.hours, d.minutes

Returns 1 year, 14 months, 10 days, 12 hours and 765 minutes.

WITH duration("P1Y2M10DT12H45M") AS d
RETURN d.years, d.months0fYear, d.days,
d.hours, d.minutes0fHour

Returns 1 year, 2 months, 10 days, 12 hours and 45 minutes.

date("2015-01-01") + duration("P1Y1M1D")

Returns a date of 2016-02-02. It is also possible to subtract durations from temporal instances.

duration("PT30S") * 10

Returns a duration of 5 minutes. It is also possible to divide a duration by a number.

Mathematical functions ☑

abs(\$expr)

The absolute value.

rand()

Returns a random number in the range from 0 (inclusive) to 1 (exclusive), [0,1). Returns a new value for each call. Also useful for selecting a subset or random ordering.

round(\$expr)

Round to the nearest integer; ceil() and floor() find the next integer up or down.

sqrt(\$expr)

The square root.

sign(\$expr)

0 if zero, -1 if negative, 1 if positive.

sin(\$expr)

Trigonometric functions also include cos(), tan(), cot(), asin(), acos(), atan(), atan2(), and haversin(). All arguments for the trigonometric functions should be in radians, if not otherwise specified.

degrees(\$expr), radians(\$expr), pi()

Converts radians into degrees; use radians() for the reverse, and pi() for π .

log10(\$expr), log(\$expr), exp(\$expr), e()

Logarithm base 10, natural logarithm, e to the power of the parameter, and the value of e.

String functions 🗗

toString(\$expression)

String representation of the expression.

replace(\$original, \$search, \$replacement)

Replace all occurrences of search with replacement. All arguments must be expressions.

substring(\$original, \$begin, \$subLength)

Get part of a string. The subLength argument is optional.

left(\$original, \$subLength),
 right(\$original, \$subLength)

The first part of a string. The last part of the string.

trim(\$original), lTrim(\$original),
 rTrim(\$original)

Trim all whitespace, or on the left or right side.

toUpper(\$original), toLower(\$original)

UPPERCASE and lowercase.

split(\$original, \$delimiter)

Split a string into a list of strings.

reverse(\$original)

Reverse a string.

size(\$string)

Calculate the number of characters in the string.

Relationship functions 🗗

type(a_relationship)

String representation of the relationship type.

startNode(a_relationship)

Start node of the relationship.

endNode(a relationship)

End node of the relationship.

id(a relationship)

The internal id of the relationship.

Aggregating functions 🗗

count(*)

The number of matching rows.

count(variable)

The number of non-null values.

count(DISTINCT variable)

All aggregating functions also take the DISTINCT operator, which removes duplicates from the values.

collect(n.property)

List from the values, ignores null.

sum(n.property)

Sum numerical values. Similar functions are avg(), min(), max().

percentileDisc(n.property, \$percentile)

Discrete percentile. Continuous percentile is percentileCont(). The percentile argument is from 0.0 to 1.0.

stDev(n.property)

Standard deviation for a sample of a population. For an entire population use stDevP().

INDEX [₹

CREATE INDEX FOR (p:Person) ON (p.name)

Create an index on the label Person and property name.

CREATE INDEX index_name FOR (p:Person) ON (p.age)

Create an index on the label Person and property age with the name index_name.

CREATE INDEX FOR (p:Person) ON (p.surname)
OPTIONS {indexProvider: 'native-btree-1.0',
indexConfig: {`spatial.cartesian.min`: [-100.0,
-100.0], `spatial.cartesian.max`: [100.0,
100.0]}}

Create an index on the label Person and property surname with the index provider native-btree-1.0 and given spatial.cartesian settings. The other index settings will have their default values.

CREATE INDEX FOR (p:Person) ON (p.name, p.age)

Create a composite index on the label Person and the properties name and age, throws an error if the index already exist.

CREATE INDEX IF NOT EXISTS FOR (p:Person) ON (p.name, p.age)

Create a composite index on the label Person and the properties name and age if it does not already exist, does nothing if it did exist.

SHOW INDEXES

List all indexes.

MATCH (n:Person) WHERE n.name = \$value
An index can be automatically used for the equality
comparison. Note that for example toLower(n.name) =
\$value will not use an index.

MATCH (n:Person)

WHERE n.name IN [\$value]

An index can automatically be used for the IN list checks.

MATCH (n:Person)

WHERE n.name = \$value and n.age = \$value2

A composite index can be automatically used for equality comparison of both properties. Note that there needs to be predicates on all properties of the composite index for it to be used.

MATCH (n:Person)

USING INDEX n:Person(name)

WHERE n.name = \$value

Index usage can be enforced when Cypher uses a suboptimal index, or more than one index should be used.

DROP INDEX index_name

Drop the index named index_name, throws an error if the index does not exist.

INDEX ☑

DROP INDEX index_name IF EXISTS

Drop the index named index_name if it exists, does nothing if it does not exist.

CONSTRAINT [₹

CREATE CONSTRAINT ON (p:Person)
ASSERT p.name IS UNIQUE

Create a unique property constraint on the label Person and property name. If any other node with that label is updated or created with a name that already exists, the write operation will fail. This constraint will create an accompanying index.

CREATE CONSTRAINT uniqueness ON (p:Person)
ASSERT p.age IS UNIQUE

Create a unique property constraint on the label Person and property age with the name uniqueness. If any other node with that label is updated or created with a age that already exists, the write operation will fail. This constraint will create an accompanying index.

CREATE CONSTRAINT ON (p:Person)

ASSERT p.surname IS UNIQUE

OPTIONS {indexProvider: 'native-btree1.0'}

Create a unique property constraint on the label Person and property surname with the index provider native-btree-1.0 for the accompanying index.

CREATE CONSTRAINT ON (p:Person)
ASSERT exists(p.name)

(*) Create a node property existence constraint on the label Person and property name, throws an error if the constraint already exists. If a node with that label is created without a name, or if the name property is removed from an existing node with the Person label, the write operation will fail.

CREATE CONSTRAINT node_exists IF NOT EXISTS ON (p:Person)

ASSERT exists(p.name)

(*) If a node property existence constraint on the label Person and property name or any constraint with the name node_exists already exist then nothing happens. If no such constraint exists, then it will be created.

CREATE CONSTRAINT ON ()-[l:LIKED]-()

ASSERT exists(l.when)

 (\star) Create a relationship property existence constraint on the type LIKED and property when. If a relationship with that type is created without a when, or if the when property is removed from an existing relationship with the LIKED type, the write operation will fail.

CREATE CONSTRAINT relationship_exists ON ()[l:LIKED]-()

ASSERT exists(l.since)

(*) Create a relationship property existence constraint on the type LIKED and property since with the name relationship_exists. If a relationship with that type is created without a since, or if the since property is removed from an existing relationship with the LIKED type, the write operation will fail.

CONSTRAINT ☐

SHOW UNIQUE CONSTRAINTS VERBOSE

List all unique constraints.

CREATE CONSTRAINT ON (p:Person)

ASSERT (p.firstname, p.surname) IS NODE KEY

(*) Create a node key constraint on the label Person and properties firstname and surname. If a node with that label is created without both firstname and surname or if the combination of the two is not unique, or if the firstname and/or surname labels on an existing node with the Person label is modified to violate these constraints, the write operation will fail.

CREATE CONSTRAINT node_key ON (p:Person)

ASSERT (p.name, p.surname) IS NODE KEY

(*) Create a node key constraint on the label Person and properties name and surname with the name node_key. If a node with that label is created without both name and surname or if the combination of the two is not unique, or if the name and/or surname labels on an existing node with the Person label is modified to violate these constraints, the write operation will fail.

CREATE CONSTRAINT node_key_with_config ON (p:Person)

ASSERT (p.name, p.age) IS NODE KEY OPTIONS {indexConfig: {`spatial.wgs-

84.min`: [-100.0, -100.0], `spatial.wgs-84.max`: [100.0, 100.0]}}

(★) Create a node key constraint on the label Person and properties name and age with the name node_key_with_config and given spatial.wgs-84 settings for the accompanying index. The other index settings will have their default values.

DROP CONSTRAINT uniqueness

Drop the constraint with the name uniqueness, throws an error if the constraint does not exist.

DROP CONSTRAINT uniqueness IF EXISTS

Drop the constraint with the name uniqueness if it exists, does nothing if it does not exist.

Performance 🗷

- 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.
- Always set an upper limit for your variable length patterns. It's possible to have a query go wild and touch all nodes in a graph by mistake.
- Return only the data you need. Avoid returning whole nodes and relationships — instead, pick the data you need and return only that.
- Use PROFILE / EXPLAIN to analyze the performance of your queries. See <u>Query Tuning</u> for more information on these and other topics, such as planner hints.

Database management 🗗

CREATE OR REPLACE DATABASE myDatabase

(★) Create a database named myDatabase. If a database with that name exists, then the existing database is deleted and a new one created.

STOP DATABASE myDatabase

(★) Stop the database myDatabase.

START DATABASE myDatabase

(★) Start the database myDatabase.

SHOW DATABASES

List all databases in the system and information about them.

SHOW DATABASE myDatabase

List information about the database myDatabase.

SHOW DEFAULT DATABASE

List information about the default database.

DROP DATABASE myDatabase IF EXISTS

(★) Delete the database myDatabase, if it exists.

(★) Role management 🗗

CREATE ROLE my_role

Create a role.

CREATE ROLE my_second_role IF NOT EXISTS AS COPY OF my role

Create a role named my_second_role, unless it already exists, as a copy of the existing my_role.

GRANT ROLE my_role, my_second_role TO alice Assign roles to a user.

REVOKE ROLE my_second_role FROM alice

Remove a specified role from a user.

SHOW ROLES

List all roles in the system.

SHOW POPULATED ROLES WITH USERS

List all roles that are assigned to at least one user in the system, and the users assigned to those roles.

DROP ROLE my_role

Delete a role.

User management 🗗

CREATE USER alice SET PASSWORD \$password

Create a new user and a password. This password must be changed on the first login.

ALTER USER alice SET PASSWORD \$password CHANGE NOT REQUIRED

Set a new password for a user. This user will not be required to change this password on the next login.

ALTER USER alice SET STATUS SUSPENDED

(★) Change the user status to suspended. Use SET STATUS ACTIVE to reactivate the user.

ALTER CURRENT USER SET PASSWORD FROM \$old TO \$new

Change the password of the logged-in user. The user will not be required to change this password on the next login.

SHOW CURRENT USER

List the currently logged-in user, their status, roles and whether they need to change their password.

(★) Status and roles are Enterprise Edition only.

SHOW USERS

List all users in the system, their status, roles and if they need to change their password.

(★) Status and roles are Enterprise Edition only.

DROP USER alice

Delete the user.

(★) Graph read privileges 🗗

GRANT TRAVERSE ON GRAPH * NODES * TO my_role

Grant traverse privilege on all nodes and all graphs to a role.

DENY READ {prop} ON GRAPH foo RELATIONSHIP Type
TO my role

Deny read privilege on a specified property, on all relationships with a specified type in a specified graph, to a role.

GRANT MATCH {*} ON DEFAULT GRAPH ELEMENTS Label
TO my_role

Grant read privilege on all properties and traverse privilege in the default graph, to a role. Here, both privileges apply to all nodes and relationships with a specified label/type in the graph.

(★) Graph write privileges 🗗

GRANT CREATE ON GRAPH * NODES Label TO my_role Grant create privilege on all nodes with a specified label in all graphs to a role.

DENY DELETE ON GRAPH neo4j TO my_role

Deny delete privilege on all nodes and relationships in a specified graph to a role.

REVOKE SET LABEL Label ON GRAPH * FROM my_role Revoke set label privilege for the specified label on all graphs to a role.

GRANT REMOVE LABEL * ON GRAPH foo TO my_role

Grant remove label privilege for all labels on a specified graph to a role.

DENY SET PROPERTY {prop} ON GRAPH foo RELATIONSHIPS Type TO my_role

Deny set property privilege on a specified property, on all relationships with a specified type in a specified graph, to a role.

GRANT MERGE {*} ON GRAPH * NODES Label TO my_role

Grant merge privilege on all properties, on all nodes with a specified label in all graphs, to a role.

REVOKE WRITE ON GRAPH * FROM my_role

Revoke write privilege on all graphs from a role.

DENY ALL GRAPH PRIVILEGES ON GRAPH foo TOmy role

Deny all graph privileges privilege on a specified graph to a role.

(★) SHOW PRIVILEGES 🗗

SHOW PRIVILEGES

List all privileges in the system, and the roles that they are assigned to.

SHOW ROLE my role PRIVILEGES

List all privileges assigned to a role.

SHOW ROLE my_role, my_second_role PRIVILEGES
List all privileges assigned to each of the multiple roles.

SHOW USER alice PRIVILEGES

List all privileges of a user, and the role that they are assigned to.

SHOW USER PRIVILEGES

Lists all privileges of the currently logged in user, and the role that they are assigned to.

SHOW PRIVILEGES AS COMMANDS

List all privileges in the system as Cypher commands.

(★) Database privileges 🗗

GRANT ACCESS ON DATABASE * TO my_role
Grant privilege to access and run queries against all
databases to a role.

GRANT START ON DATABASE * TO my_role Grant privilege to start all databases to a role.

GRANT STOP ON DATABASE * TO my_role
Grant privilege to stop all databases to a role.

GRANT CREATE INDEX ON DATABASE foo TO my_role Grant privilege to create indexes on a specified database to a role.

GRANT DROP INDEX ON DATABASE foo TO my_role
Grant privilege to drop indexes on a specified database to a role.

GRANT SHOW INDEX ON DATABASE * TO my_role
Grant privilege to show indexes on all databases to a role.

DENY INDEX MANAGEMENT ON DATABASE bar TO my_role

Deny privilege to create and drop indexes on a specified database to a role.

GRANT CREATE CONSTRAINT ON DATABASE * TO my role

Grant privilege to create constraints on all databases to a role.

DENY DROP CONSTRAINT ON DATABASE * TO my_role Deny privilege to drop constraints on all databases to a role.

DENY SHOW CONSTRAINT ON DATABASE foo TO my_role
Deny privilege to show constraints on a specified database
to a role.

REVOKE CONSTRAINT ON DATABASE * FROM my_role
Revoke granted and denied privileges to create and drop
constraints on all databases from a role.

GRANT CREATE NEW LABELS ON DATABASE * TO my role

Grant privilege to create new labels on all databases to a role.

DENY CREATE NEW TYPES ON DATABASE foo TO my_role

Deny privilege to create new relationship types on a specified database to a role.

REVOKE GRANT CREATE NEW PROPERTY NAMES ON DATABASE bar FROM my_role

Revoke the grant privilege to create new property names on a specified database from a role.

GRANT NAME MANAGEMENT ON DEFAULT DATABASE TO my_role

Grant privilege to create labels, relationship types, and property names on default database to a role.

(★) Role management privileges 🖸

GRANT CREATE ROLE ON DBMS TO my_role Grant the privilege to create roles to a role.

GRANT DROP ROLE ON DBMS TO my_role Grant the privilege to delete roles to a role.

DENY ASSIGN ROLE ON DBMS TO my_role

Deny the privilege to assign roles to users to a role.

DENY REMOVE ROLE ON DBMS TO my_role

Deny the privilege to remove roles from users to a role.

REVOKE DENY SHOW ROLE ON DBMS FROM my_role
Revoke the denied privilege to show roles from a role.

GRANT ROLE MANAGEMENT ON DBMS TO my_role Grant all privileges to manage roles to a role.

(★) User management privileges 🗗

GRANT CREATE USER ON DBMS TO my_role Grant the privilege to create users to a role.

GRANT DROP USER ON DBMS TO my_role Grant the privilege to delete users to a role.

DENY ALTER USER ON DBMS TO my_role Deny the privilege to alter users to a role.

REVOKE SET PASSWORDS ON DBMS FROM my_role
Revoke the granted and denied privileges to alter users'
passwords from a role.

REVOKE GRANT SET USER STATUS ON DBMS FROM my role

Revoke the granted privilege to alter the account status of users from a role.

REVOKE DENY SHOW USER ON DBMS FROM my_role
Revoke the denied privilege to show users from a role.

GRANT USER MANAGEMENT ON DBMS TO my_role Grant all privileges to manage users to a role.

(★) Database management privileges 🗗

GRANT CREATE DATABASE ON DBMS TO my_role Grant the privilege to create databases to a role.

REVOKE DENY DROP DATABASE ON DBMS FROM my_role Revoke the denied privilege to delete databases from a role.

DENY DATABASE MANAGEMENT ON DBMS TO my_role
Deny all privileges to manage database to a role.

(★) Database privileges 🗗

GRANT ALL ON DATABASE baz TO my_role

Grant privilege to access, create and drop indexes and constraints, create new labels, types and property names on a specified database to a role.

GRANT SHOW TRANSACTION (*) ON DATABASE foo TO
my_role

Grant privilege to list transactions and queries from all users on a specified database to a role.

DENY TERMINATE TRANSACTION (user1, user2) ON DATABASES * TO my_role

Deny privilege to kill transactions and queries from user1 and user2 on all databases to a role.

REVOKE GRANT TRANSACTION MANAGEMENT ON DEFAULT DATABASE FROM my_role

Revoke the granted privilege to list and kill transactions and queries from all users on the default database from a role.

(★) Privilege management privileges 🗗

GRANT SHOW PRIVILEGE ON DBMS TO my_role Grant the privilege to show privileges to a role.

DENY ASSIGN PRIVILEGE ON DBMS TO my_role
Deny the privilege to assign privileges to roles to a role.

REVOKE GRANT REMOVE PRIVILEGE ON DBMS FROM my_role

Revoke the granted privilege to remove privileges from roles from a role.

REVOKE PRIVILEGE MANAGEMENT ON DBMS FROM my_role

Revoke all granted and denied privileges for manage privileges from a role.

(★) DBMS privileges 🗗

GRANT ALL ON DBMS TO my role

Grant privilege to perform all role management, user management, database management and privilege management to a role.



(★) Functionality available in Neo4j Enterprise Edition.