Neo4j
the world's leading graph database

Cypher is the declarative query language for Neo4j, the world's leading graph database.

Key principles and capabilities of Cypher are as follows:

- Cypher matches patterns of nodes and relationship in the graph, to extract information or modify the data.
- Cypher has the concept of identifiers which denote named, bound elements and parameters.
- Cypher can create, update, and remove nodes, relationships, and properties.

You can try cypher snippets live in the Neo4j Console at console.neo4j.org or read the full Cypher documentation at docs.neo4j.org.

Note: `{value}` denotes either literals, for ad hoc Cypher queries; or parameters, which is the best practice for applications.

Syntax

Read Query Structure

```
START
[MATCH]
[WHERE]
RETURN [ORDER BY] [SKIP] [LIMIT]
```

START

```
START n=node(*)
```
Start from all nodes.

```
START n=node({ids})
```
Start from one or more nodes specified by id.

```
START n=node({id1}), m=node({id2})
```
Multiple starting points.

```
START n=node:nodeIndexName(key={value})
```
Query the index with an exact query. Use `node_auto_index` for the automatic index.

MATCH

```
MATCH (n)-->(m)
```
Any pattern can be used in `MATCH` except the ones containing property maps.

WHERE

```
WHERE n.property <> {value}
```
Use a predicate to filter.

RETURN

```
RETURN *
```
Return the value of all identifiers.

```
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 {skip_number}
```
Skip a number of results.

```
LIMIT {limit_number}
```
Limit the number of results.

```
SKIP {skip_number} LIMIT {limit_number}
```
Skip results at the top and limit the number of results.

WITH

```
START user=node:nodeIndexName(name = {name})
MATCH (user)-[:FRIEND]-(friend)
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 identifiers to carry over to the next part.

Patterns

```
(n)-->(m)
```
A relationship from `n` to `m` exists.

```
(n)--(m)
```
A relationship from `n` to `m` or from `m` to `n` exists.

```
(m)<-[:KNOWS]-(n)
```
A relationship from `n` to `m` of type `KNOWS` exists.

```
(n)-[:KNOWS|LOVES]->(m)
```
A relationship from `n` to `m` of type `KNOWS` or `LOVES` exists.

```
(n)-[r]->(m)
```
Bind an identifier to the relationship.

```
(n)-[r?]->(m)
```
Optional relationship.

```
(n)-[*1..5]->(m)
```
Variable length paths.

```
(n)-[*]->(m)
```
Any depth.

```
(n)-[:KNOWS]->(m {property: {value}})
```
Match or set properties in `CREATE` or `CREATE UNIQUE` clauses.

Write-Only Query Structure

```
CREATE [UNIQUE]*
[SET|DELETE|FOREACH]*
[RETURN [ORDER BY] [SKIP] [LIMIT]]
```

Read-Write Query Structure

```
START
[MATCH]
[WHERE]
[CREATE [UNIQUE]]*
[SET|DELETE|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.

```
CREATE n = {collectionOfMaps}
```
Create nodes with the given properties.

```
CREATE (n)-[r:KNOWS]->(m)
```
Create a relationship with the given type and direction; bind an identifier to it.

```
CREATE (n)-[:LOVES {since: {value}}]->(m)
```
Create a relationship with the given type, direction, and properties.

CREATE UNIQUE

```
CREATE UNIQUE
    (n)-[:KNOWS]->(m {property: {value}})
```
Match pattern or create it if it does not exist. The pattern can not include any optional parts.

SET

```
SET n.property={value}
```
Update or create a property.

```
SET n={map}
```
Set all properties. This will remove any existing properties.

DELETE

```
DELETE n, r
```
Delete a node and a relationship.

```
DELETE n.property
```
Delete a property.

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.
- Avoid using optional relationships. Cypher has to employ a much slower pattern matcher when you use optional relationships.
- Always set an upper limit for your variable length relationships. It's easy 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.

Neo4j Cypher Refcard 1.9

Operators

Mathematical	`+`, `-`, `*`, `/`, `%`
Comparison	`=`, `<>`, `<`, `>`, `<=`, `>=`
Boolean	`AND`, `OR`, `NOT`
String	`+`
Collection	`+`, `IN`
Regular Expression	`=~`
Property	`?`, `!`

Predicates

`n.property <> {value}`
Use comparison operators.

`HAS(n.property) AND n.property = {value}`
Use boolean operators to combine predicates.

`HAS(n.property)`
Use functions.

`identifier IS NULL`
Check if something is `null`.

`n.property? = {value}`
Defaults to `true` if the property does not exist.

`n.property! = {value}`
Defaults to `false` if the property does not exist.

`n.property =~ {regex}`
Regular expression.

`(n)-[:KNOWS]->(m)`
Make sure the pattern has at least one match.

`n.property IN [{value1}, {value2}]`
Check if an element exists in a collection.

Predicate Functions

`ALL(x IN collection WHERE HAS(x.property))`
Returns `true` if the predicate is `true` for all elements of the collection.

`ANY(x IN collection WHERE HAS(x.property))`
Returns `true` if the predicate is `true` for at least one element of the collection.

`NONE(x IN collection WHERE HAS(x.property))`
Returns `true` if the predicate is `false` for all elements of the collection.

`SINGLE(x IN collection WHERE HAS(x.property))`
Returns `true` if the predicate is `true` for exactly one element in the collection.

Scalar Functions

`LENGTH(collection)`
Length of the collection.

`TYPE(a_relationship)`
String representation of the relationship type.

`COALESCE(n.property?, {defaultValue})`
The first non-`null` expression.

`HEAD(collection)`
The first element of the collection.

`LAST(collection)`
The last element of the collection.

`TIMESTAMP()`
Milliseconds since midnight, January 1, 1970 UTC.

`ID(node_or_relationship)`
The internal id of the relationship or node.

Mathematical Functions

`ABS({numerical_expression})`
The absolute value.

`ROUND({numerical_expression})`
Round to the nearest integer.

`SQRT({numerical_expression})`
The square root.

`SIGN({numerical_expression})`
`0` if zero, `-1` if negative, `1` if positive.

Aggregation

`COUNT(*)`
The number of matching rows.

`COUNT(identifier)`
The number of non-`null` values.

`COUNT(DISTINCT identifier)`
All aggregation functions also take the `DISTINCT` modifier, which removes duplicates from the values

`SUM(n.property)`
Sum numerical values.

`AVG(n.property)`
Calculates the average.

`MAX(n.property)`
Maximum numerical value.

`MIN(n.property)`
Minimum numerical value.

`COLLECT(n.property?)`
Collection from the values, ignores `null`.

`PERCENTILE_DISC(n.property, {percentile})`
Discrete percentile. The `percentile` argument is from `0.0` to `1.0`.

`PERCENTILE_CONT(n.property, {percentile})`
Continuous percentile.

Collection Functions

`NODES(path)`
The nodes in the path.

`RELATIONSHIPS(path)`
The relationships in the path.

`EXTRACT(x IN collection: x.prop)`
A collection of the value of the expression for each element in the collection.

`FILTER(x IN coll: x.prop <> {value})`
A collection of the elements where the predicate is `true`.

`TAIL(collection)`
All but the first element of the collection.

`RANGE({begin}, {end}, {step})`
Create a range of numbers. The `step` argument is optional.

`REDUCE(str = "", n IN coll : str + n.prop)`
Evaluate expression for each element in the collection, accumulate the results.

`FOREACH (n IN coll : SET n.marked = true)`
Execute a mutating operation for each element in a collection.

String Functions

`STR({expression})`
String representation of the expression.

`REPLACE({original}, {search}, {replacement})`
Replace all occurrences of `search` with `replacement`. All arguments are be expressions.

`SUBSTRING({original}, {begin}, {sub_length})`
Get part of a string. The `substring_length` argument is optional.

`LEFT({original}, {substring_length})`
The first part of a string.

`RIGHT({original}, {substring_length})`
The last part of a string.

`LTRIM({original})`
No whitespace on the left side.

`RTRIM({original})`
No whitespace on the right side.

`TRIM({original})`
No whitespace on the left or right side.

`LOWER({original})`
Lowercase.

`UPPER({original})`
Uppercase.