



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, labels, and properties.
- Cypher manages indexes and constraints.

You can try Cypher snippets live in the Neo4j Console at [console.neo4j.org](https://console.neo4j.org) or read the full Cypher documentation at [docs.neo4j.org](https://docs.neo4j.org). For live graph models using Cypher check out [GraphGist](#).

Note: {value} denotes either literals, for ad hoc Cypher queries; or parameters, which is the best practice for applications. Neo4j properties can be strings, numbers, booleans or arrays thereof. Cypher also supports maps and collections.

WHERE

WHERE n.property <> {value}

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

Syntax

Read Query Structure

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

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.

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.

RETURN count(\*)

The number of matching rows. See Aggregation for more.

## WITH

```
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 identifiers 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
```

You can also use ORDER BY, SKIP, LIMIT with WITH.

## 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.

## UNWIND

```
MATCH p = shortestPath( (lucy:Person
{name:"Lucy Liu"})-[:ACTED_IN*]-
(bacon:Person {name:"Kevin Bacon"}) )
UNWIND nodes(p) as n
```

Transform a collection back into individual rows.

## Write-Only Query Structure

```
(CREATE [UNIQUE] | MERGE)*
[SET|DELETE|REMOVE|FOREACH]*
[RETURN [ORDER BY] [SKIP] [LIMIT]]
```

## Read-Write Query Structure

```
[MATCH WHERE]
[OPTIONAL MATCH WHERE]
[WITH [ORDER BY] [SKIP] [LIMIT]]
(CREATE [UNIQUE] | 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.

```
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.

## 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 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 subgraphs attached to the node.

SET
<b>SET n.property = {value}, n.property2 = {value2}</b>
Update or create a property.
<b>SET n={map}</b>
Set all properties. This will remove any existing properties.
<b>SET n:Person</b>
Adds a label <b>Person</b> to a node.

DELETE
<b>DELETE n, r</b>
Delete a node and a relationship.

REMOVE
<b>REMOVE n:Person</b>
Remove a label from n.
<b>REMOVE n.property</b>
Remove a property.

INDEX
<b>CREATE INDEX ON :Person(name)</b>
Create an index on the label <b>Person</b> and property <b>name</b> .
<b>MATCH (n:Person) WHERE n.name = {value}</b>
An index can be automatically used for the equality comparison. Note that for example <b>lower(n.name) = {value}</b> will not use an index.
<b>MATCH (n:Person) USING INDEX n:Person(name) WHERE n.name = {value}</b>
Index usage can be enforced, when Cypher uses a suboptimal index or more than one index should be used.
<b>DROP INDEX ON :Person(name)</b>
Drop the index on the label <b>Person</b> and property <b>name</b> .

CONSTRAINT
<b>CREATE CONSTRAINT ON (p:Person) ASSERT p.name IS UNIQUE</b>
Create a unique constraint on the label <b>Person</b> and property <b>name</b> . If any other node with that label is updated or created with a <b>name</b> that already exists, the write operation will fail. This constraint will create an accompanying index.
<b>DROP CONSTRAINT ON (p:Person) ASSERT p.name IS UNIQUE</b>
Drop the unique constraint and index on the label <b>Person</b> and property <b>name</b> .

Operators	
Mathematical	+, -, *, /, %, ^
Comparison	=, <>, <, >, <=, >=
Boolean	AND, OR, XOR, NOT
String	+
Collection	+, IN, [x], [x .. y]
Regular Expression	=~

NULL
<ul style="list-style-type: none"><li>• NULL is used to represent missing/undefined values.</li><li>• NULL is not equal to NULL. Not knowing two values does not imply that they are the same value. So the expression <code>NULL = NULL</code> yields <code>NULL</code> and not <code>TRUE</code>. To check if an expressoin is NULL, use <code>IS NULL</code>.</li><li>• Arithmetic expressions, comparisons and function calls (except <code>coalesce</code>) will return <code>NULL</code> if any argument is <code>NULL</code>.</li><li>• Missing elements like a property that doesn't exist or accessing elements that don't exist in a collection yields <code>NULL</code>.</li><li>• In <code>OPTIONAL MATCH</code> clauses, <code>NULLs</code> will be used for missing parts of the pattern.</li></ul>

Patterns
<code>(n) - -&gt; (m)</code>
A relationship from <code>n</code> to <code>m</code> exists.
<code>(n:Person)</code>
Matches nodes with the label <code>Person</code> .
<code>(n:Person:Swedish)</code>
Matches nodes which have both <code>Person</code> and <code>Swedish</code> labels.
<code>(n:Person {name: {value}})</code>
Matches nodes with the declared properties.
<code>(n:Person) - -&gt; (m)</code>
Node <code>n</code> labeled <code>Person</code> has a relationship to <code>m</code> .
<code>(n) - - (m)</code>
A relationship in any direction between <code>n</code> and <code>m</code> .
<code>(m) &lt;- [ :KNOWS ] - (n)</code>
A relationship from <code>n</code> to <code>m</code> of type <code>KNOWS</code> exists.
<code>(n) - [ :KNOWS   LOVES ] -&gt; (m)</code>
A relationship from <code>n</code> to <code>m</code> of type <code>KNOWS</code> or <code>LOVES</code> exists.
<code>(n) - [ r ] -&gt; (m)</code>
Bind an identifier to the relationship.
<code>(n) - [*1..5] -&gt; (m)</code>
Variable length paths.
<code>(n) - [*] -&gt; (m)</code>
Any depth. See the performance tips.
<code>(n) - [ :KNOWS ] -&gt; (m {property: {value}})</code>
Match or set properties in <code>MATCH</code> , <code>CREATE</code> , <code>CREATE UNIQUE</code> or <code>MERGE</code> clauses.
<code>shortestPath((n1:Person) - [*..6] - (n2:Person))</code>
Find a single shortest path.
<code>allShortestPaths((n1:Person) - -&gt; (n2:Person))</code>
Find all shortest paths.

## 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 label and property.

```
SET n:Spouse:Parent:Employee
```

Add label(s) to a node.

```
MATCH (n:Person)
```

Matches nodes labeled as `Person`.

```
MATCH (n:Person)
WHERE n.name = {value}
```

Matches nodes labeled `Person` with the given name.

```
WHERE (n:Person)
```

Checks existence of label on node.

```
labels(n)
```

Labels of the node.

```
REMOVE n:Person
```

Remove label from node.

## Collections

```
['a','b','c'] AS coll
```

Literal collections are declared in square brackets.

```
length({coll}) AS len, {coll}[0] AS
value
```

Collections can be passed in as parameters.

```
range({first_num},{last_num},{step}) AS
coll
```

Range creates a collection of numbers (step is optional), other functions returning collections are: `labels`, `nodes`, `relationships`, `rels`, `filter`, `extract`.

```
MATCH (a)-[r:KNOWS*]->()
RETURN r AS rels
```

Relationship identifiers of a variable length path contain a collection of relationships.

```
RETURN matchedNode.coll[0] AS value,
       length(matchedNode.coll) AS len
```

Properties can be arrays/collections of strings, numbers or booleans.

```
coll[{idx}] AS value,
coll[{start_idx}..{end_idx}] AS slice
```

Collection 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.

## Maps

```
{name:'Alice', age:38,
 address:{city:'London',
 residential:true}}
```

Literal maps are declared in curly braces much like property maps. Nested maps and collections are supported.

```
MERGE (p:Person {name: {map}.name})
ON CREATE SET p={map}
```

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

```
RETURN matchedNode AS map
```

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.

## 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.

Predicates
<code>n.property &lt;&gt; {value}</code>
Use comparison operators.
<code>has(n.property)</code>
Use functions.
<code>n.number &gt;= 1 AND n.number &lt;= 10</code>
Use boolean operators to combine predicates.
<code>n:Person</code>
Check for node labels.
<code>identifier IS NULL</code>
Check if something is NULL.
<code>NOT has(n.property) OR n.property = {value}</code>
Either property does not exist or predicate is TRUE.
<code>n.property = {value}</code>
Non-existing property returns NULL, which is not equal to anything.
<code>n.property =~ "Tob.*"</code>
Regular expression.
<code>(n) - [:KNOWS] -&gt;(m)</code>
Make sure the pattern has at least one match.
<code>NOT (n) - [:KNOWS] -&gt;(m)</code>
Exclude matches to <code>(n) - [:KNOWS] -&gt;(m)</code> from the result.
<code>n.property IN [{value1}, {value2}]</code>
Check if an element exists in a collection.

Collection Predicates
<code>all(x IN coll WHERE has(x.property))</code>
Returns <code>true</code> if the predicate is TRUE for all elements of the collection.
<code>any(x IN coll WHERE has(x.property))</code>
Returns <code>true</code> if the predicate is TRUE for at least one element of the collection.
<code>none(x IN coll WHERE has(x.property))</code>
Returns TRUE if the predicate is FALSE for all elements of the collection.
<code>single(x IN coll WHERE has(x.property))</code>
Returns TRUE if the predicate is TRUE for exactly one element in the collection.

Functions
<code>coalesce(n.property, {defaultValue})</code>
The first non-NULL expression.
<code>timestamp()</code>
Milliseconds since midnight, January 1, 1970 UTC.
<code>id(node_or_relationship)</code>
The internal id of the relationship or node.
<code>toInt({expr})</code>
Converts the given input in an integer if possible; otherwise it returns NULL.
<code>toFloat({expr})</code>
Converts the given input in a floating point number if possible; otherwise it returns NULL.

Path Functions
<code>length(path)</code>
The length of the path.
<code>nodes(path)</code>
The nodes in the path as a collection.
<code>relationships(path)</code>
The relationships in the path as a collection.
<code>MATCH path=(n) - -&gt;(m)</code> <code>RETURN extract(x IN nodes(path)   x.prop)</code>
Assign a path and process its nodes.
<code>MATCH path = (begin) -[*]-&gt; (end)</code> <code>FOREACH</code> <code>  (n IN rels(path)   SET n.marked = TRUE)</code>
Execute a mutating operation for each relationship of a path.

Collection Functions
length({coll})
Length of the collection.
head({coll}), last({coll}), tail({coll})
head returns the first, last the last element of the collection. tail the remainder of the collection. All return null for an empty collection.
[x IN coll WHERE x.prop <> {value}   x.prop]
Combination of filter and extract in a concise notation.
extract(x IN coll   x.prop)
A collection of the value of the expression for each element in the original collection.
filter(x IN coll WHERE x.prop <> {value})
A filtered collection of the elements where the predicate is TRUE.
reduce(s = "", x IN coll   s + x.prop)
Evaluate expression for each element in the collection, accumulate the results.
FOREACH (value IN coll   CREATE (:Person {name:value}))
Execute a mutating operation for each element in a collection.

Mathematical Functions
abs({expr})
The absolute value.
rand()
A random value. Returns a new value for each call. Also useful for selecting 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 cos, tan, cot, asin, acos, atan, atan2, haversin.
degrees({expr}), radians({expr}), pi()
Converts radians into degrees, use radians for the reverse. pi for π.
log10({expr}), log({expr}), exp({expr}), e()
Logarithm base 10, natural logarithm, e to the power of the parameter. Value of e.

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 sub_length argument is optional.
left({original}, {sub_length}), right({original}, {sub_length})
The first part of a string. The last part of the string.
trim({original}), ltrim({original}), rtrim({original})
Trim all whitespace, or on left or right side.
upper({original}), lower({original})
UPPERCASE and lowercase.
split({original}, {delimiter})
Split a string into a collection of strings.

Aggregation
<code>count(*)</code>
The number of matching rows.
<code>count(identifier)</code>
The number of non-NULL values.
<code>count(DISTINCT identifier)</code>
All aggregation functions also take the <b>DISTINCT</b> modifier, which removes duplicates from the values.
<code>collect(n.property)</code>
Collection from the values, ignores NULL.
<code>sum(n.property)</code>
Sum numerical values. Similar functions are <code>avg</code> , <code>min</code> , <code>max</code> .
<code>percentileDisc(n.property, {percentile})</code>
Discrete percentile. Continuous percentile is <code>percentileCont</code> . The <code>percentile</code> argument is from <code>0.0</code> to <code>1.0</code> .
<code>stdev(n.property)</code>
Standard deviation for a sample of a population. For an entire population use <code>stdevp</code> .

CASE
<pre> CASE n.eyes   WHEN 'blue' THEN 1   WHEN 'brown' THEN 2   ELSE 3 END </pre>
Return <b>THEN</b> value from the matching <b>WHEN</b> value. The <b>ELSE</b> value is optional, and substituted for <b>NULL</b> if missing.
<pre> CASE   WHEN n.eyes = 'blue' THEN 1   WHEN n.age &lt; 40 THEN 2   ELSE 3 END </pre>
Return <b>THEN</b> value from the first <b>WHEN</b> predicate evaluating to <b>TRUE</b> . Predicates are evaluated in order.

Upgrading
With Neo4j 2.0 several Cypher features in version 1.9 have been deprecated or removed.
<ul style="list-style-type: none"> <li>• <b>START</b> is optional.</li> <li>• <b>MERGE</b> will take <b>CREATE UNIQUE</b>'s role for the unique creation of patterns. Note that they are not the same, though.</li> <li>• Optional relationships are handled by <b>OPTIONAL MATCH</b>, not question marks.</li> <li>• Non-existing properties return <b>NULL</b>, <code>n.prop?</code> and <code>n.prop!</code> have been removed.</li> <li>• The separator for collection functions changed from <code>:</code> to <code> </code>.</li> <li>• Paths are no longer collections, use <code>nodes(path)</code> or <code>rels(path)</code>.</li> <li>• Parentheses around nodes in patterns are no longer optional.</li> <li>• <code>CREATE a={property:'value'}</code> has been removed.</li> <li>• Use <b>REMOVE</b> to remove properties.</li> <li>• Parameters for index-keys and nodes in patterns are no longer allowed.</li> <li>• To still use the older syntax, prepend your Cypher statement with <code>CYPHER 1.9</code>.</li> </ul>



## 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.

## 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.

## 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 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.