

## 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 in the [Neo4j Manual](#). For live graph models using Cypher check out [GraphGist](#).

The Cypher Refcard is also [available in PDF format](#).

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 lists.

## 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.

```
WHERE m.name = "Alice"
```

Force the planner to use a label scan to solve the query (for manual performance tuning).

**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.

**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.

```
UNWIND {listOfMaps} AS properties CREATE (n) SET n = properties
```

Create nodes with the given properties.

```
CREATE (n)-[: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.

**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.

**REMOVE**

```
REMOVE n:Person
```

Remove a label from `n`.

```
REMOVE n.property
```

Remove a property.

**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 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 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
```

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.

**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)-[:LOVES]->(b)
```

`MERGE` finds or creates a relationship between the nodes.

```
MATCH (a:Person {name: {value1}})
MERGE
    (a)-[:KNOWS]->(b:Person {name: {value3}})
```

`MERGE` finds or creates subgraphs attached to the node.

**DELETE**

```
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.

**FOREACH**

```
FOREACH (r IN rels(path) |
    SET r.marked = TRUE)
```

Execute a mutating operation for each relationship of a path.

```
FOREACH (value IN coll |
    CREATE (:Person {name: value}))
```

Execute a mutating operation for each element in a list.

**CALL**

```
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`.

**START**

```
START n = node:nodeIndexName(key = {value})
```

Query the index named `nodeIndexName` with an exact query. Use `node_auto_index` for the automatic index. Note that other uses of `START` have been removed as of Cypher 2.2.

**INDEX**

```
CREATE INDEX ON :Person(name)
```

Create an index on the label `Person` and property `name`.

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

An index can be automatically used for the equality comparison. Note that for example `lower(n.name) = {value}` will not use an index.

```
MATCH (n:Person) WHERE n.name IN [{value}]
```

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

```
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 ON :Person(name)
```

Drop the index on the label `Person` and property `name`.

**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.

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

Drop the unique constraint and index on the label `Person` and property `name`.

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

Create a node property existence constraint on the label `Person` and property `name`. 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.

```
DROP CONSTRAINT ON (p:Person)
    ASSERT exists(p.name)
```

Drop the node property existence constraint on the label `Person` and property `name`.

```
CREATE CONSTRAINT ON ()-[l:LIKED]-()
    ASSERT exists(l.when)
```

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.

```
DROP CONSTRAINT ON ()-[l:LIKED]-()
    ASSERT exists(l.when)
```

Drop the relationship property existence constraint on the type `LIKED` and property `when`.

**Import**

```
LOAD CSV FROM
'http://neo4j.com/docs/3.0.3/cypher-refcard/csv/artists.csv' AS line
CREATE (:Artist {name: line[1], year:
toInt(line[2])})
```

Load data from a CSV file and create nodes.

```
LOAD CSV WITH HEADERS FROM
'http://neo4j.com/docs/3.0.3/cypher-refcard/csv/artists-with-headers.csv' AS line
CREATE (:Artist {name: line.Name, year:
toInt(line.Year)})
```

Load CSV data which has headers.

```
LOAD CSV FROM
'http://neo4j.com/docs/3.0.3/cypher-refcard/csv/artists-fieldterminator.csv'
AS line FIELDTERMINATOR ';'
CREATE (:Artist {name: line[1], year:
toInt(line[2])})
```

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

Operators	
Mathematical	+, -, *, /, %, ^
Comparison	=, <>, <, >, <=, >=
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.

**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.

Patterns
<code>(n:Person)</code> Node with <code>Person</code> label.
<code>(n:Person:Swedish)</code> Node with both <code>Person</code> and <code>Swedish</code> labels.
<code>(n:Person {name: {value}})</code> Node with the declared properties.
<code>(n)--&gt;(m)</code> Relationship from <code>n</code> to <code>m</code> .
<code>(n)--(m)</code> Relationship in any direction between <code>n</code> and <code>m</code> .
<code>(n:Person)--&gt;(m)</code> Node <code>n</code> labeled <code>Person</code> with relationship to <code>m</code> .
<code>(m)-[:KNOWS]-(n)</code> Relationship of type <code>KNOWS</code> from <code>n</code> to <code>m</code> .
<code>(n)-[:KNOWS LOVES]-&gt;(m)</code> Relationship of type <code>KNOWS</code> or of type <code>LOVES</code> from <code>n</code> to <code>m</code> .
<code>(n)-[r]-&gt;(m)</code> Bind the relationship to variable <code>r</code> .
<code>(n)-[*1..5]-&gt;(m)</code> Variable length path of between 1 and 5 relationships from <code>n</code> to <code>m</code> .
<code>(n)-[*]-&gt;(m)</code> Variable length path of any number of relationships from <code>n</code> to <code>m</code> . (Please see the performance tips.)
<code>(n)-[:KNOWS]-&gt;(m {property: {value}})</code> A relationship of type <code>KNOWS</code> from a node <code>n</code> to a node <code>m</code> with the declared property.
<code>shortestPath((n1:Person)-[*..6]-(n2:Person))</code> Find a single shortest path.
<code>allShortestPaths((n1:Person)-[*..6]-&gt;(n2:Person))</code> Find all shortest paths.
<code>size((n)--&gt;()-&gt;())</code> Count the paths matching the pattern.

Maps
<code>{name: "Alice", age: 38, address: {city: "London", residential: true}}</code> Literal maps are declared in curly braces much like property maps. Nested maps and list are supported.
<code>MERGE (p:Person {name: {map}.name}) ON CREATE SET p = {map}</code> Maps can be passed in as parameters and used as map or by accessing keys.
<code>MATCH (matchedNode:Person) RETURN matchedNode</code> Nodes and relationships are returned as maps of their data.
<code>map.name, map.age, map.children[0]</code> Map entries can be accessed by their keys. Invalid keys result in an error.

Predicates
<code>n.property &lt;&gt; {value}</code> Use comparison operators.
<code>exists(n.property)</code> Use functions.
<code>n.number &gt;= 1 AND n.number &lt;= 10</code> Use boolean operators to combine predicates.
<code>1 &lt;= n.number &lt;= 10</code> Use chained operators to combine predicates.
<code>n:Person</code> Check for node labels.
<code>variable IS NULL</code> Check if something is <code>NULL</code> .
<code>NOT exists(n.property) OR n.property = {value}</code> Either property does not exist or predicate is <code>TRUE</code> .
<code>n.property = {value}</code> Non-existing property returns <code>NULL</code> , which is not equal to anything.
<code>n["property"] = {value}</code> Properties may also be accessed using a dynamically computed property name.
<code>n.property STARTS WITH "Tob" OR n.property ENDS WITH "n" OR n.property CONTAINS "goodie"</code> String matching.
<code>n.property =~ "Tob.*"</code> String regular expression matching.
<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 list.

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

Relationship Functions
<code>type(a_relationship)</code> String representation of the relationship type.
<code>startNode(a_relationship)</code> Start node of the relationship.
<code>endNode(a_relationship)</code> End node of the relationship.
<code>id(a_relationship)</code> The internal id of the relationship.

List Predicates
<code>all(x IN coll WHERE exists(x.property))</code> Returns <code>true</code> if the predicate is <code>true</code> for all elements of the list.
<code>any(x IN coll WHERE exists(x.property))</code> Returns <code>true</code> if the predicate is <code>true</code> for at least one element of the list.
<code>none(x IN coll WHERE exists(x.property))</code> Returns <code>true</code> if the predicate is <code>false</code> for all elements of the list.
<code>single(x IN coll WHERE exists(x.property))</code> Returns <code>true</code> if the predicate is <code>true</code> for exactly one element in the list.

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

Path Functions
<code>length(path)</code> The number of relationships in the path.
<code>nodes(path)</code> The nodes in the path as a list.
<code>relationships(path)</code> The relationships in the path as a list.
<code>extract(x IN nodes(path)   x.prop)</code> Extract properties from the nodes in a path.

Mathematical Functions
<code>abs({expr})</code> The absolute value.
<code>rand()</code> 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 subset or random ordering.
<code>round({expr})</code> Round to the nearest integer, <code>ceil</code> and <code>floor</code> find the next integer up or down.
<code>sqrt({expr})</code> The square root.
<code>sign({expr})</code> 0 if zero, -1 if negative, 1 if positive.
<code>sin({expr})</code> Trigonometric functions, also <code>cos</code> , <code>tan</code> , <code>cot</code> , <code>asin</code> , <code>acos</code> , <code>atan</code> , <code>atan2</code> , <code>haversin</code> . All arguments for the trigonometric functions should be in radians, if not otherwise specified.
<code>degrees({expr}), radians({expr}), pi()</code> Converts radians into degrees, use <code>radians</code> for the reverse. <code>pi</code> for $\pi$ .
<code>log10({expr}), log({expr}), exp({expr}), e()</code> Logarithm base 10, natural logarithm, <code>e</code> to the power of the parameter. Value of <code>e</code> .

String Functions
<code>toString({expression})</code> String representation of the expression.
<code>replace({original}, {search}, {replacement})</code> Replace all occurrences of <code>search</code> with <code>replacement</code> . All arguments are be expressions.
<code>substring({original}, {begin}, {subLength})</code> Get part of a string. The <code>subLength</code> argument is optional.
<code>left({original}, {subLength}), right({original}, {subLength})</code> The first part of a string. The last part of the string.
<code>trim({original}), ltrim({original}), rtrim({original})</code> Trim all whitespace, or on left or right side.
<code>upper({original}), lower({original})</code> UPPERCASE and lowercase.
<code>split({original}, {delimiter})</code> Split a string into a list of strings.
<code>reverse({original})</code> Reverse a string.
<code>length({string})</code> Calculate the number of characters in the string.

Labels
<code>CREATE (n:Person {name: {value}})</code> Create a node with label and property.
<code>MERGE (n:Person {name: {value}})</code> Matches or creates unique node(s) with label and property.
<code>SET n:Spouse:Parent:Employee</code> Add label(s) to a node.
<code>MATCH (n:Person)</code> Matches nodes labeled <code>Person</code> .
<code>MATCH (n:Person) WHERE n.name = {value}</code> Matches nodes labeled <code>Person</code> with the given <code>name</code> .
<code>WHERE (n:Person)</code> Checks existence of label on node.
<code>labels(n)</code> Labels of the node.
<code>REMOVE n:Person</code> Remove label from node.

Lists
<code>["a", "b", "c"] AS coll</code> Literal lists are declared in square brackets.
<code>size({coll}) AS len, {coll}[0] AS value</code> Lists can be passed in as parameters.
<code>range({firstNum}, {lastNum}, {step}) AS coll</code> Range creates a list of numbers ( <code>step</code> is optional), other functions returning list are: <code>labels</code> , <code>nodes</code> , <code>relationships</code> , <code>rels</code> , <code>filter</code> , <code>extract</code> .
<code>MATCH (a)-[r:KNOWS*]-&gt;(c) RETURN r AS rels</code> Relationship variables of a variable length path contain a list of relationships.
<code>RETURN matchedNode.coll[0] AS value, size(matchedNode.coll) AS len</code> Properties can be lists of strings, numbers or booleans.
<code>coll[{idx}] AS value, coll[{startIdx}..{endIdx}] AS slice</code> List elements can be accessed with <code>idx</code> subscripts in square brackets. Invalid indexes return <code>NULL</code> . Slices can be retrieved with intervals from <code>start_idx</code> to <code>end_idx</code> each of which can be omitted or negative. Out of range elements are ignored.
<code>UNWIND {names} AS name MATCH (n {name: name}) RETURN avg(n.age)</code> With <code>UNWIND</code> , you can transform any list back into individual rows. The example matches all names from a list of names.

List Expressions
<code>size({coll})</code> Number of elements in the list.
<code>head({coll}), last({coll}), tail({coll})</code> <code>head</code> returns the first, <code>last</code> the last element of the list. <code>tail</code> returns all but the first element. All return <code>NULL</code> for an empty list.
<code>[x IN coll WHERE x.prop &lt;&gt; {value}   x.prop]</code> Combination of <code>filter</code> and <code>extract</code> in a concise notation.
<code>extract(x IN coll   x.prop)</code> A list of the value of the expression for each element in the original list.
<code>filter(x IN coll WHERE x.prop &lt;&gt; {value})</code> A filtered list of the elements where the predicate is <code>TRUE</code> .
<code>reduce(s = "", x IN coll   s + x.prop)</code> Evaluate expression for each element in the list, accumulate the results.

Aggregation
<code>count(*)</code> The number of matching rows.
<code>count(variable)</code> The number of non- <code>NULL</code> values.
<code>count(DISTINCT variable)</code> All aggregation functions also take the <code>DISTINCT</code> modifier, which removes duplicates from the values.
<code>collect(n.property)</code> List from the values, ignores <code>NULL</code> .
<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 0.0 to 1.0.
<code>stdev(n.property)</code> Standard deviation for a sample of a population. For an entire population use <code>stdevp</code> .

Performance
<ul style="list-style-type: none"> <li>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.</li> <li>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.</li> <li>Return only the data you need. Avoid returning whole nodes and relationships — instead, pick the data you need and return only that.</li> <li>Use <code>PROFILE</code> / <code>EXPLAIN</code> to analyze the performance of your queries. See <a href="#">Query Tuning</a> for more information.</li> </ul>