



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 collections.

Syntax

Read Query Structure
<pre>[MATCH WHERE] [OPTIONAL MATCH WHERE] [WITH [ORDER BY] [SKIP] [LIMIT]] RETURN [ORDER BY] [SKIP] [LIMIT]</pre>
MATCH
<pre>MATCH (n:Person)-[:KNOWS]-&gt;(m:Person) WHERE n.name = "Alice"</pre> Node patterns can contain labels and properties.
<pre>MATCH (n)--&gt;(m)</pre> Any pattern can be used in MATCH.
<pre>MATCH (n {name: "Alice"})--&gt;(m)</pre> Patterns with node properties.
<pre>MATCH p = (n)--&gt;(m)</pre> Assign a path to p.
<pre>OPTIONAL MATCH (n)-[r]-&gt;(m)</pre> Optional pattern, NULLs will be used for missing parts.
<pre>WHERE m.name = "Alice"</pre> Force the planner to use a label scan to solve the query (for manual performance tuning).

WHERE
<pre>WHERE n.property &lt;&gt; {value}</pre> 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
<pre>(CREATE [UNIQUE]   MERGE)* [SET DELETE REMOVE FOREACH]* [RETURN [ORDER BY] [SKIP] [LIMIT]]</pre>

Read-Write Query Structure
<pre>[MATCH WHERE] [OPTIONAL MATCH WHERE] [WITH [ORDER BY] [SKIP] [LIMIT]] (CREATE [UNIQUE]   MERGE)* [SET DELETE REMOVE FOREACH]* [RETURN [ORDER BY] [SKIP] [LIMIT]]</pre>

CREATE
<pre>CREATE (n {name: {value}})</pre> Create a node with the given properties.
<pre>CREATE (n {map})</pre> Create a node with the given properties.
<pre>CREATE (n {collectionOfMaps})</pre> Create nodes with the given properties.
<pre>CREATE (n)-[r:KNOWS]-&gt;(m)</pre> Create a relationship with the given type and direction; bind an identifier to it.
<pre>CREATE (n)-[:LOVES {since: {value}}]-&gt;(m)</pre> Create a relationship with the given type, direction, and properties.

MERGE
<pre>MERGE (n:Person {name: {value}}) ON CREATE SET n.created = timestamp() ON MATCH SET     n.counter = coalesce(n.counter, 0) + 1,     n.accessTime = timestamp()</pre> Match pattern or create it if it does not exist. Use ON CREATE and ON MATCH for conditional updates.
<pre>MATCH (a:Person {name: {value1}}),       (b:Person {name: {value2}}) MERGE (a)-[r:LOVES]-&gt;(b)</pre> MERGE finds or creates a relationship between the nodes.
<pre>MATCH (a:Person {name: {value1}}) MERGE     (a)-[r:KNOWS]-&gt;(b:Person {name: {value3}})</pre> MERGE finds or creates subgraphs attached to the node.

RETURN
<pre>RETURN *</pre> Return the value of all identifiers.
<pre>RETURN n AS columnName</pre> Use alias for result column name.
<pre>RETURN DISTINCT n</pre> Return unique rows.
<pre>ORDER BY n.property</pre> Sort the result.
<pre>ORDER BY n.property DESC</pre> Sort the result in descending order.
<pre>SKIP {skipNumber}</pre> Skip a number of results.
<pre>LIMIT {limitNumber}</pre> Limit the number of results.
<pre>SKIP {skipNumber} LIMIT {limitNumber}</pre> Skip results at the top and limit the number of results.
<pre>RETURN count(*)</pre> The number of matching rows. See Aggregation for more.

WITH
<pre>MATCH (user)-[:FRIEND]-(friend) WHERE user.name = {name} WITH user, count(friend) AS friends WHERE friends &gt; 10 RETURN user</pre> 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.
<pre>MATCH (user)-[:FRIEND]-(friend) WITH user, count(friend) AS friends ORDER BY friends DESC SKIP 1 LIMIT 3 RETURN user</pre> You can also use ORDER BY, SKIP, LIMIT with WITH.

UNION
<pre>MATCH (a)-[:KNOWS]-&gt;(b) RETURN b.name UNION MATCH (a)-[:LOVES]-&gt;(b) RETURN b.name</pre> Returns the distinct union of all query results. Result column types and names have to match.
<pre>MATCH (a)-[:KNOWS]-&gt;(b) RETURN b.name UNION ALL MATCH (a)-[:LOVES]-&gt;(b) RETURN b.name</pre> Returns the union of all query results, including duplicated rows.

SET
<pre>SET n.property1 = {value1},     n.property2 = {value2}</pre> Update or create a property.
<pre>SET n = {map}</pre> Set all properties. This will remove any existing properties.
<pre>SET n += {map}</pre> Add and update properties, while keeping existing ones.
<pre>SET n:Person</pre> Adds a label Person to a node.

DELETE
<pre>DELETE n, r</pre> Delete a node and a relationship.
<pre>DETACH DELETE n</pre> Delete a node and all relationships connected to it.
<pre>MATCH (n) DETACH DELETE n</pre> Delete all nodes and relationships from the database.

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

FOREACH
<pre>FOREACH (r IN rels(path)       SET r.marked = TRUE)</pre> Execute a mutating operation for each relationship of a path.
<pre>FOREACH (value IN coll       CREATE (:Person {name:value}))</pre> Execute a mutating operation for each element in a collection.

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

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

CONSTRAINT
<pre>CREATE CONSTRAINT ON (p:Person)     ASSERT p.name IS UNIQUE</pre> 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.
<pre>DROP CONSTRAINT ON (p:Person)     ASSERT p.name IS UNIQUE</pre> Drop the unique constraint and index on the label Person and property name.
<pre>CREATE CONSTRAINT ON (p:Person)     ASSERT exists(p.name)</pre> 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.
<pre>DROP CONSTRAINT ON (p:Person)     ASSERT exists(p.name)</pre> Drop the node property existence constraint on the label Person and property name.
<pre>CREATE CONSTRAINT ON ()-[l:LIKED]-()     ASSERT exists(l.when)</pre> 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.
<pre>DROP CONSTRAINT ON ()-[l:LIKED]-()     ASSERT exists(l.when)</pre> Drop the relationship property existence constraint on the type LIKED and property when.

Import
<pre>LOAD CSV FROM 'http://neo4j.com/docs/2.3.3/cypher-refcard/csv/artists.csv' AS line CREATE (:Artist {name: line[1], year: toInt(line[2])})</pre> Load data from a CSV file and create nodes.
<pre>LOAD CSV WITH HEADERS FROM 'http://neo4j.com/docs/2.3.3/cypher-refcard/csv/artists-with-headers.csv' AS line CREATE (:Artist {name: line.Name, year: toInt(line.Year)})</pre> Load CSV data which has headers.
<pre>LOAD CSV FROM 'http://neo4j.com/docs/2.3.3/cypher-refcard/csv/artists-fieldterminator.csv' AS line FIELDTERMINATOR ',' CREATE (:Artist {name: line[1], year: toInt(line[2])})</pre> Use a different field terminator, not the default which is a comma (with no whitespace around it).

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 NULL = NULL yields NULL and not TRUE. To check if an expression is NULL, use IS NULL.</li><li>• Arithmetic expressions, comparisons and function calls (except coalesce) will return NULL if any argument is NULL.</li><li>• An attempt to access a missing element in a collection or a property that doesn’t exist yields NULL.</li><li>• In OPTIONAL MATCH clauses, NULLs will be used for missing parts of the pattern.</li></ul>

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



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)&lt;-[: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 identifier <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.

Collections
<code>["a", "b", "c"] AS coll</code> Literal collections are declared in square brackets.
<code>size({coll}) AS len, {coll}[0] AS value</code> Collections can be passed in as parameters.
<code>range({firstNum}, {lastNum}, {step}) AS coll</code> Range creates a collection of numbers ( <code>step</code> is optional), other functions returning collections 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;()</code> <code>RETURN r AS rels</code> Relationship identifiers of a variable length path contain a collection of relationships.
<code>RETURN matchedNode.coll[0] AS value, size(matchedNode.coll) AS len</code> Properties can be arrays/collections of strings, numbers or booleans.
<code>coll[{idx}] AS value, coll[{startIndex}..{endIdx}] AS slice</code> Collection 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</code> <code>MATCH (n {name: name})</code> <code>RETURN avg(n.age)</code> With <code>UNWIND</code> , you can transform any collection back into individual rows. The example matches all names from a list of names.

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 collections are supported.
<code>MERGE (p:Person {name: {map}.name})</code> <code>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)</code> <code>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.

CASE
<code>CASE n.eyes</code> <code>WHEN "blue" THEN 1</code> <code>WHEN "brown" THEN 2</code> <code>ELSE 3</code> <code>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</code> <code>WHEN n.eyes = "blue" THEN 1</code> <code>WHEN n.age &lt; 40 THEN 2</code> <code>ELSE 3</code> <code>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.

Collection 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 collection.
<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 collection.
<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 collection.
<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 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(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 collection 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 collection.
<code>relationships(path)</code> The relationships in the path as a collection.
<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> A random number between 0 and 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> <code>0</code> if zero, <code>-1</code> if negative, <code>1</code> 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> .
<code>degrees({expr})</code> , <code>radians({expr})</code> , <code>pi()</code> Converts radians into degrees, use <code>radians</code> for the reverse. <code>pi</code> for $\pi$ .
<code>log10({expr})</code> , <code>log({expr})</code> , <code>exp({expr})</code> , <code>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})</code> , <code>right({original}, {subLength})</code> The first part of a string. The last part of the string.
<code>trim({original})</code> , <code>ltrim({original})</code> , <code>rtrim({original})</code> Trim all whitespace, or on left or right side.
<code>upper({original})</code> , <code>lower({original})</code> UPPERCASE and lowercase.
<code>split({original}, {delimiter})</code> Split a string into a collection of strings.
<code>reverse({original})</code> Reverse a string.
<code>length({string})</code> Calculate the number of characters in the string.

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>identifier 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 collection.

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

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 <code>DISTINCT</code> modifier, which removes duplicates from the values.
<code>collect(n.property)</code> Collection 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 <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> .

START
<code>START n = node:nodeIndexName(key = {value})</code> Query the index named <code>nodeIndexName</code> with an exact query. Use <code>node_auto_index</code> for the automatic index. Note that other uses of <code>START</code> have been removed as of Cypher 2.2.

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

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>