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 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, `NULL`s 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)-[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.

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)-[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.

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/3.0.0/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/3.0.0/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/3.0.0/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, `NULL`s 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

`(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.

`(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 `r`.

`(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`. (Please see the performance tips.)

`(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.

Maps

```
{name: "Alice", age: 38,
 address: {city: "London", residential: true}}
```
Literal maps are declared in curly braces much like property maps. Nested maps and list 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.

```
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

`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 property does not exist or 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 "Tob" OR
n.property ENDS WITH "n" OR
n.property CONTAINS "goodie"
```
String matching.

`n.property =~ "Tob.*"`
String regular expression matching.

`(n)-[:KNOWS]->(m)`
Make sure 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.

CASE

```
CASE n.eyes
 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.

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.

List Predicates

`all(x IN coll WHERE exists(x.property))`
Returns `true` if the predicate is `true` for all elements of the list.

`any(x IN coll WHERE exists(x.property))`
Returns `true` if the predicate is `true` for at least one element of the list.

`none(x IN coll WHERE exists(x.property))`
Returns `true` if the predicate is `false` for all elements of the list.

`single(x IN coll WHERE exists(x.property))`
Returns `true` if the predicate is `true` for exactly one element in the list.

Functions

`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.

`toInt({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`.

`keys({expr})`
Returns a list of string representations for the property names of a node, relationship, or map.

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.

`extract(x IN nodes(path) | x.prop)`
Extract properties from the nodes in a path.

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 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`. 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. `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

`toString({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}, {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 left or right side.

`upper({original}), lower({original})`
UPPERCASE and lowercase.

`split({original}, {delimiter})`
Split a string into a list of strings.

`reverse({original})`
Reverse a string.

`length({string})`
Calculate the number of characters in the string.

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 `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.

Lists

`["a", "b", "c"] AS coll`
Literal lists are declared in square brackets.

`size({coll}) AS len, {coll}[0] AS value`
Lists can be passed in as parameters.

`range({firstNum}, {lastNum}, {step}) AS coll`
Range creates a list of numbers (`step` is optional), other functions returning list are: `labels`, `nodes`, `relationships`, `rels`, `filter`, `extract`.

```
MATCH (a)-[r:KNOWS*]->()
RETURN r AS rels
```
Relationship variables of a variable length path contain a list of relationships.

```
RETURN matchedNode.coll[0] AS value,
       size(matchedNode.coll) AS len
```
Properties can be lists of strings, numbers or booleans.

```
coll[{idx}] AS value,
coll[{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`, you can transform any list back into individual rows. The example matches all names from a list of names.

List Expressions

`size({coll})`
Number of elements in the list.

`head({coll}), last({coll}), tail({coll})`
`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 coll WHERE x.prop <> {value} | x.prop]`
Combination of filter and extract in a concise notation.

`extract(x IN coll | x.prop)`
A list of the value of the expression for each element in the original list.

`filter(x IN coll WHERE x.prop <> {value})`
A filtered list of the elements where the predicate is `TRUE`.

`reduce(s = "", x IN coll | s + x.prop)`
Evaluate expression for each element in the list, accumulate the results.

Aggregation

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

`count(variable)`
The number of non-`NULL` values.

`count(DISTINCT variable)`
All aggregation functions also take the `DISTINCT` modifier, 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`.

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.
- Use `PROFILE` / `EXPLAIN` to analyze the performance of your queries. See Query Tuning for more information.