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 relationships in the graph, to extract information or modify the data.
- Cypher has the concept of variables 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 Developer Manual. For live graph models using Cypher check out GraphGist.

The Cypher Refcard is also available in PDF format.

Note: `{value}` denotes either literals or maps, used for ad hoc Cypher queries. The usage of parameters is recommended in applications, and are denoted by `$value`. Neo4j properties can be strings, numbers, booleans or arrays thereof. Cypher also supports maps and lists.

Legend

Read
Write
General
Functions
Schema
Performance

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'})-->(n)
```
Patterns with node properties.

```
MATCH p = (n)-->(n)
```
Assign a path to `p`.

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 Aggregating Functions 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
```
`ORDER BY`, `SKIP`, and `LIMIT` can also be used 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,
```

Operators

General	`DISTINCT`, `.`, `[]`
Mathematical	`+`, `-`, `*`, `/`, `%`, `^`
Comparison	`=`, `<>`, `<`, `>`, `<=`, `>=`, `IS NULL`, `IS NOT NULL`
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.

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.

```
()-[r {name: $value}]-()
```
Matches relationships with the declared properties.

```
(n)-->(m)
```
Relationship from `n` to `m`.

```
(n)--(n)
```
Relationship in any direction between `n` and `n`.

```
(n:Person)-->(m)
```
Node `n` labeled `Person` with relationship to `m`.

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

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

```
(n)-[r]->(n)
```
Bind the relationship to variable `r`.

```
(n)-[*1..5]->(m)
```
Variable length path of between 1 and 5 relationships from `n` to `m`.

```
(n)-[*]->(n)
```
Variable length path of any number of relationships from

Assign a path to `p`.

```
OPTIONAL MATCH (n)-[r]->(m)
```
Optional pattern: `null`s will be used for missing parts.

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] {vbar} MERGE)*
[SET{vbar}DELETE{vbar}REMOVE{vbar}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.

Import

```
LOAD CSV FROM
'https://neo4j.com/docs/cypher-
refcard/3.2/csv/artists.csv' AS line
CREATE (:Artist {name: line[1], year: toInt(line[2])})
```

```
ON MATCH SET
  n.counter = coalesce(n.counter, 0) + 1,
  n.accessTime = timestamp()
```
Match a 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.

REMOVE

```
REMOVE n:Person
```
Remove a label from `n`.

```
REMOVE n.property
```
Remove a property.

FOREACH

```
FOREACH (r IN relationships(path) |
  SET r.marked = true)
```

```
  SET r.marked = true)
```
Execute a mutating operation for each relationship in 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`.

```
(n)-[*]->(m)
```
Variable length path of any number of relationships from `n` to `m`. (See Performance section.)

```
(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.

Lists

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

```
size($list) AS len, $list[0] AS value
```
Lists can be passed in as parameters.

```
range($firstNum, $lastNum, $step) AS list
```
`range()` creates a list of numbers (`step` is optional), other functions returning lists are: `labels()`, `nodes()`, `relationships()`, `filter()`, `extract()`.

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

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

```
list[$idx] AS value,
list[$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`, any list can be transformed back into individual rows. The example matches all names from a list of names.

```
MATCH (a)
RETURN [(a)-->(b) WHERE b.name = 'Bob' | b.age]
```
Pattern comprehensions may be used to do a custom projection from a match directly into a list.

```
MATCH (person)
RETURN person { .name, .age}
```
Map projections may be easily constructed from nodes, relationships and other map values.