Neo4j Cypher Cheatsheet

A quick reference guide to Neo4j basics, data modeling concepts, and essential Cypher query language syntax and commands.

Neo4j Basics & Concepts

Core Concepts

Nodes: Represent entities. Can have properties (key-value pairs) and labels (typed groups).
Relationships: Connect nodes. Always directed, have a type, and can have properties.
Properties: Key-value pairs stored on nodes or relationships. Values can be primitives (string, number, boolean) or arrays of primitives.
Labels: Typed groups for nodes. A node can have multiple labels. Used for indexing and constraints.
Relationship Types: Typed connections between nodes. A relationship must have exactly one type.
Schema-Free: Neo4j is schema-flexible. Properties and labels/types are not strictly defined at creation but are typically enforced with constraints.
Graph Structure: Data is stored as connected nodes and relationships, optimizing for traversals.
Cypher: Neo4j's declarative query language for working with the graph.

Node Syntax

()	An anonymous node.
(n)	A node bound to variable `n`.
(:Label)	A node with a specific label.
(n:Label)	A node bound to variable `n` with a specific label.
(n:Label:Label2)	A node with multiple labels.
({key: 'value'})	A node with properties.
(:Label {key: 'value', num: 123})	A node with a label and properties.
(n:Label {key: 'value'})	A node bound to `n` with a label and properties.

Relationship Syntax

-->	A directed relationship.
<--	A directed relationship (reverse direction).
--	An undirected relationship (not common in queries, primarily for representation).
-[:TYPE]->	A relationship with a specific type.
-[r:TYPE]->	A relationship bound to variable `r` with a specific type.
-[:TYPE|TYPE2]->	A relationship with one of multiple types.
-[:TYPE {key: 'value'}]->	A relationship with a type and properties.
-[:TYPE*]->	Variable length relationship (1 or more).
-[:TYPE*2..5]->	Variable length relationship (2 to 5 hops).

Graph Patterns

`(a)-->(b)`: Simple pattern matching two nodes `a` and `b` connected by a directed relationship.

`(a:Person)-[:KNOWS]->(b:Person)`: Pattern matching two `Person` nodes connected by a `KNOWS` relationship.

`(a:Movie {title: 'The Matrix'})<--(:ACTED_IN)--(p:Person)`: Pattern matching a `Movie` node with a specific title connected by an incoming `ACTED_IN` relationship to a `Person` node `p`.

`(p:Person)-[r:WORKS_AT]->(c:Company) WHERE r.startDate < 2000`: Pattern matching a `Person` and `Company` connected by a `WORKS_AT` relationship, filtered by a relationship property.

`(a)-[*1..5]->(b)`: Variable length path of 1 to 5 hops between `a` and `b` (type optional).

`(a)-[:TYPE*]->(b)`: Variable length path of 1 or more hops of a specific type.

Multiple Patterns:

```
MATCH (p:Person)-[:LIVES_IN]->(c:City),
      (p)-[:WORKS_AT]->(co:Company)
RETURN p, c, co
```

Optional Match:

```
OPTIONAL MATCH (p:Person)-[:LIVES_IN]->
(c:City)
RETURN p, c
```

Returns persons even if they don't have a `:LIVES_IN` relationship.

Essential Cypher Queries

Create Data

Create a node:

```
CREATE (p:Person {name: 'Alice', age: 30})
RETURN p
```

Create multiple nodes:

```
CREATE (:Movie {title: 'Inception'}),
       (:Movie {title: 'Interstellar'})
```

Create a relationship:

```
MATCH (a:Person {name: 'Alice'}), (b:Person {name: 'Bob'})
CREATE (a)-[:KNOWS {since: 2015}]->(b)
RETURN a, b
```

Create a node and relationship simultaneously:

```
CREATE (p:Person {name: 'Charlie'})-[:LIVES_IN]->(c:City {name:
'Paris'})
RETURN p, c
```

Create nodes with multiple labels:

```
CREATE (e:Employee:Person {name: 'Dave'})
RETURN e
```

Create a relationship with multiple properties:

```
MATCH (p:Person {name: 'Alice'}), (m:Movie {title: 'Inception'})
CREATE (p)-[:ACTED_IN {roles: ['Dom Cobb'], year: 2010}]->(m)
RETURN p, m
```

Create a variable-length relationship (rarely used in CREATE, more in MATCH):

```
// Conceptually possible, but usually involves multiple simple
CREATEs
// or leveraging MERGE/MATCH to find existing nodes.
```

Match Data

Match all nodes:

```
MATCH (n)
RETURN n LIMIT 10
```

Match nodes with a specific label:

```
MATCH (p:Person)
RETURN p LIMIT 10
```

Match a node by property:

```
MATCH (p:Person {name: 'Alice'})
RETURN p
```

Match a node by property using WHERE:

```
MATCH (p:Person)
WHERE p.age > 25
RETURN p.name, p.age
```

Match a specific relationship type:

```
MATCH (a)-[:KNOWS]->(b)
RETURN a, b LIMIT 10
```

Match relationships with specific properties:

```
MATCH (a)-[r:KNOWS]->(b)
WHERE r.since = 2015
RETURN a.name, b.name, r.since
```

Match paths:

```
MATCH path = (a:Person)-[:KNOWS*..2]->(b:Person)
WHERE a.name = 'Alice'
RETURN path
```

Match nodes connected by any relationship type:

```
MATCH (a)-[r]->(b)
RETURN type(r), a.name, b.name LIMIT 10
```

Merge Data

Merge a node (create if not exists, find if exists):

```
MERGE (c:City {name: 'London'})
RETURN c
```

Merge a relationship (create if pattern not exists):

```
MATCH (a:Person {name: 'Alice'}), (c:City {name: 'London'})
MERGE (a)-[:LIVES_IN]->(c)
RETURN a, c
```

ON CREATE / ON MATCH clauses:
Set properties only if the node/relationship was created:

```
MERGE (p:Person {name: 'Bob'})
ON CREATE SET p.born = 1990
RETURN p
```

ON CREATE / ON MATCH clauses:
Set properties only if the node/relationship was matched:

```
MERGE (p:Person {name: 'Bob'})
ON MATCH SET p.lastSeen = timestamp()
RETURN p
```

Combined ON CREATE and ON MATCH:

```
MERGE (p:Person {name: 'Charlie'})
ON CREATE SET p.created = timestamp(), p.status = 'New'
ON MATCH SET p.updated = timestamp(), p.status = 'Existing'
RETURN p
```

Merge complex patterns:

```
MERGE (u:User {userId: 123})-[:BOUGHT]->(p:Product {productId: 456})
RETURN u, p
```

Using MERGE on relationship with properties:

```
MATCH (p1:Person {name: 'Alice'}), (p2:Person {name: 'Bob'})
MERGE (p1)-[k:KNOWS]->(p2)
ON CREATE SET k.since = 2023
ON MATCH SET k.strength = k.strength + 1
RETURN p1, p2, k
```

Merging multiple paths:

```
MERGE (a:Team {name: 'Red'}), (b:Team {name: 'Blue'})
MERGE (a)-[:RIVALRY]-(b)
RETURN a, b
```

Update & Delete Data

SET - Set or update properties/labels.

Set a property:

```
MATCH (p:Person {name: 'Alice'})
SET p.age = 31
RETURN p
```

Set multiple properties:

```
MATCH (p:Person {name: 'Bob'})
SET p += {city: 'New York', zip: '10001'}
RETURN p
```

Set a label:

```
MATCH (p:Person {name: 'Charlie'})
SET p:Employee
RETURN p
```

Update relationship properties:

```
MATCH (:Person {name: 'Alice'})-[k:KNOWS]->(:Person {name: 'Bob'})
SET k.strength = 10
RETURN k
```

REMOVE - Remove properties/labels.

Remove a property:

```
MATCH (p:Person {name: 'Alice'})
REMOVE p.age
RETURN p
```

Remove multiple properties:

```
MATCH (p:Person {name: 'Bob'})
REMOVE p.city, p.zip
RETURN p
```

Remove a label:

```
MATCH (p:Person:Employee {name: 'Charlie'})
REMOVE p:Employee
RETURN p
```

Remove relationship properties:

```
MATCH (:Person {name: 'Alice'})-[k:KNOWS]->(:Person {name: 'Bob'})
REMOVE k.since
RETURN k
```

DELETE - Delete nodes and relationships.

Delete relationships (nodes remain):

```
MATCH (:Person {name: 'Alice'})-[k:KNOWS]->(:Person {name: 'Bob'})
DELETE k
```

Delete nodes based on a condition:

```
MATCH (p:Person)
WHERE p.age > 60
DETACH DELETE p
```

DETACH DELETE - Delete nodes and their relationships.

Delete a node and its relationships:

```
MATCH (p:Person {name: 'Alice'})
DETACH DELETE p
```

Delete all nodes and relationships:

```
MATCH (n)
DETACH DELETE n
```

Caution: This empties the database!

Return & Ordering

RETURN - Specify what to output.

Return nodes and relationships:

```
MATCH (p:Person)-[r:KNOWS]->
(f:Person)
RETURN p, r, f
```

Return specific properties:

```
MATCH (p:Person)
RETURN p.name AS Name, p.age
```

Use `AS` for aliases.

Return distinct results:

```
MATCH (p:Person)-[:ACTED_IN]->
(m:Movie)
RETURN DISTINCT m.title
```

Return aggregated values (COUNT, SUM, AVG, MIN, MAX, COLLECT, etc.):

```
MATCH (p:Person)
RETURN count(p) AS totalPeople
```

SKIP & LIMIT - Pagination.

Skip first N results:

```
MATCH (p:Person)
RETURN p.name
ORDER BY p.name
SKIP 10
```

Combine SKIP and LIMIT:

```
MATCH (p:Person)
RETURN p.name
ORDER BY p.name
SKIP 10 LIMIT 5
```

ORDER BY - Sort results.

Order by node property (ascending):

```
MATCH (p:Person)
RETURN p.name, p.age
ORDER BY p.age
```

Order by node property (descending):

```
MATCH (p:Person)
RETURN p.name, p.age
ORDER BY p.age DESC
```

Order by multiple properties:

```
MATCH (p:Person)
RETURN p.name, p.age,
p.city
ORDER BY p.city, p.age
DESC
```

Ordering on aggregates:

```
MATCH (m:Movie)<-
[:ACTED_IN]-(p:Person)
RETURN m.title, count(p)
AS actorCount
ORDER BY actorCount DESC
```

Limit results to N:

```
MATCH (p:Person)
RETURN p.name
ORDER BY p.name
LIMIT 5
```

Return paths:

```
MATCH path = (a:Person)-
[:KNOWS*..2]->(b:Person)
RETURN path
```

Predicates & Functions

Comparison: `=` , `<>` , `<` , `>` , `<=` , `>=`

```
WHERE p.age > 30
```

Boolean: `AND` , `OR` , `XOR` , `NOT`

```
WHERE p.age > 30 AND
p.city = 'London'
```

Regular Expressions: `=~`

```
WHERE p.name =~ '(?i)alice.*'
```

(Case-insensitive starts with 'alice')

String Predicates: `STARTS WITH` , `ENDS WITH` , `CONTAINS`

```
WHERE p.name STARTS
WITH 'A'
```

List Predicates: `IN`

```
WHERE p.city IN ['Paris', 'London']
```

Null Check: `IS NULL` , `IS NOT NULL`

```
WHERE p.age IS NOT
NULL
```

Existence Check: `EXISTS()`

```
WHERE EXISTS(p.email)
```

Relationship type: `type(r)`

```
MATCH (a)-[r]->(b)
WHERE type(r) =
'FRIEND'
RETURN a, b
```

Node labels: `LABELS(n)`

```
MATCH (n)
WHERE 'Person' IN labels(n)
RETURN n
```

Path length: `length(path)`

```
MATCH path=(a)-[*]->
(b)
WHERE length(path) >
3
RETURN path
```

Functions: `keys()` , `properties()` , `size()` , `id()` , `timestamp()` , `datetime()` , etc.

Get properties:

```
MATCH (p:Person {name: 'Alice'})
RETURN properties(p)
```

Get keys:

```
MATCH (p:Person
{name: 'Alice'})
RETURN keys(p)
```

Get ID:

```
MATCH (p:Person
{name: 'Alice'})
RETURN id(p)
```

Aggregating Functions: `count()` , `sum()` , `avg()` , `min()` , `max()` , `collect()` , `stdev()` , `percentileCont()` , etc.

Count all nodes:

```
MATCH (n)
RETURN count(n)
```

Collect properties into a list:

```
MATCH (p:Person)
RETURN
collect(p.name) AS
names
```

Constraints & Indexes

Constraints enforce rules on data, like uniqueness.
Indexes speed up lookups on node/relationship properties.

Create a uniqueness constraint (also creates an index):

```
CREATE CONSTRAINT ON (p:Person) ASSERT p.email IS UNIQUE
```

Create a node property existence constraint:

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

Create a relationship property existence constraint:

```
CREATE CONSTRAINT ON ()-[r:KNOWS]-() ASSERT EXISTS(r.since)
```

Create a composite uniqueness constraint:

```
CREATE CONSTRAINT ON (p:Product) ASSERT (p.id, p.version) IS
UNIQUE
```

Create a B-tree index on a node property:

```
CREATE INDEX ON :Movie(title)
```

Create a B-tree index on a relationship property:

```
CREATE INDEX ON :LIKES(rating)
```

Drop a constraint:

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

Drop an index:

```
DROP INDEX ON :Movie(title)
```

Show constraints:

```
SHOW CONSTRAINTS
```

Show indexes:

```
SHOW INDEXES
```