

Neo4j cheat sheet v2.2

Manipulating data

Adding nodes and relationships

```
CREATE (:Typename {attribute1: 'value1', attribute2: 'value2', ...});
CREATE (...node designation...) -[:RelTypename]-> (...node designation...);
```

Note: you always have to create *directional* relationships, but when querying, you don't have to specify a direction.

Be careful not to “overcreate” duplicate nodes! Wrong:

```
create (:Bard {name:'Redbeard',level:5,hitPoints:27});
create (:Bard {name:'Redbeard'})-[:wields]->(:Weapon {name:'Broadsword'});
```

This creates duplicate Bard objects. Instead, do this:

Right:

```
create (:Bard {name:'Redbeard',level:5,hitPoints:27});
match (r:Bard {name:'Redbeard'})
    create (r)-[:wields]->(:Weapon {name:'Broadsword'});
```

Deleting nodes or relationships

```
match (a:...)
    DELETE a;
```

(or DETACH DELETE to nuke nodes that have relationships, along with their relationships)

Looking at the “schema”

Here's some really useful queries that show you what all the labels and properties are in the database:

```
match (a) return labels(a), count(*) order by count(*) desc;
match (a) return keys(a), count(*) order by count(*) desc;
match (a) return labels(a), keys(a), count(*) order by count(*) desc;
match (a)-[r]->(b) return distinct type(r);
match (a)-[r]->(b) return distinct labels(a), type(r), labels(b);
match ()-[r]->() return distinct type(r), keys(r);
```

Querying

All of the following techniques can be combined.

Return every node

```
match (a)
    return a;
```

Return nodes of given type

```
match (a:SomeTypeName)
  return a;
```

Return nodes based on properties

nodes with specific properties (at all)

```
match (a)
  where a.pref is not null
  return a.creds;
```

nodes with specific property values

```
match (a {pref:"CPSC", numCreds:4})
  return a;
```

partial string matching

```
match (a:Person)
  where a.name contains "ephe"
  return a.name;
```

```
match (a:Person)
  where a.name starts with "Z"
  return a.name;
```

Return specific properties

```
match (a)
  return a.name, a.age;
```

(Can also use `distinct` here.)

The “COALESCE” operator

```
match (a)
  return coalesce(a.name, a.first, a.last) as name;
```

(this sets “name” to the first of whichever of those three fields actually exists)

Query based on relationships

```
match (a)-->(b)
  return a.name, b.name;
```

Regardless of direction

```
match (a)--(b)
  return a.name, b.name;
```

Based on relationship type(s)

```
match (a)-[:SomeRelType]->(b)
  return a.name, b.name;
```

```
match (a)-[:SomeRelType|SomeOtherRelType]->(b)
  return a.name, b.name;
```

Based on relationship properties

```
match (a)-[{year:2020}]->(b)
  return a.name, b.name;
```

Combining many query options

```
match (a:Character {species:"human"})-[r:appearsWith {ep:"V"}]->(b:Character {species:"wookie"})
  return a.name, b.name, r.loc;
```

Multiple “hops”

```
match (a:Character {name:"Yoda"})-[:touches*1..2]->(b:Character)
  return distinct b.name;
```

Count the hops:

```
match path = (a:Character {name:"Yoda"})-[:touches*1..3]->(b:Character)
  return distinct b.name, length(path) as numHops order by numHops;
```

Sorting (“order by”)

```
match (a)-[:parent]->(b:Professor)
  return b.name, count(*) order by count(*) desc;
```

Aggregation (like “group by”)

In a move I heartily applaud, Neo4j makes “group by” sort of automatic when you include an aggregation operator like `count()` or `sum()`:

```
match (a:Course)
  return a.pref, avg(a.creds);
```

```
match (a:Character)
  return a.species, count(*);
```

```
match (a:Person)-[:parent]->()
  return max(a.age);
```

Skip & Limit

For paging, like `limit/offset` in SQL:

```
match (a)<-[:mentions]->(b)
  return b.name order by b.name
  skip 5 limit 2;
```

Python support

```
$ pip install neo4j
```

```
from neo4j import GraphDatabase
neoconn = GraphDatabase.driver("bolt://localhost:7687",
    auth=("neo4j","davies4ever"))
session = neoconn.session()
results = session.run("...neo4j query...")
results.data()
```