

Contents

[Graph Database Fundamentals 1](#_Toc169022995)

[1. Nodes 1](#_Toc169022996)

[2. Labels 1](#_Toc169022997)

[3. Relationships 1](#_Toc169022998)

[4. Properties 2](#_Toc169022999)

[NEO4J BASICS 6](#_Toc169023000)

[Index free adjacency 6](#_Toc169023001)

[ACID 7](#_Toc169023002)

[Clusters 7](#_Toc169023003)

[Graph engine 8](#_Toc169023004)

[Bolt 8](#_Toc169023005)

[Cypher language 8](#_Toc169023006)

[Match 9](#_Toc169023007)

[Properties 13](#_Toc169023008)

[Relationships 15](#_Toc169023009)

[Where 19](#_Toc169023010)

[Aggregation in Cypher 23](#_Toc169023011)

[Create 26](#_Toc169023012)

[Deleting nodes and relationships 29](#_Toc169023013)

[Merge 30](#_Toc169023014)

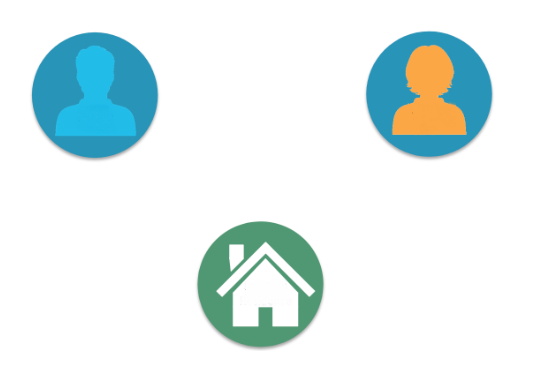
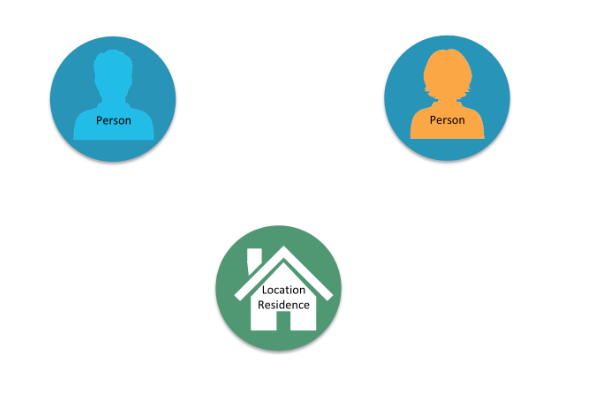
[Managing indexes 32](#_Toc169023015)

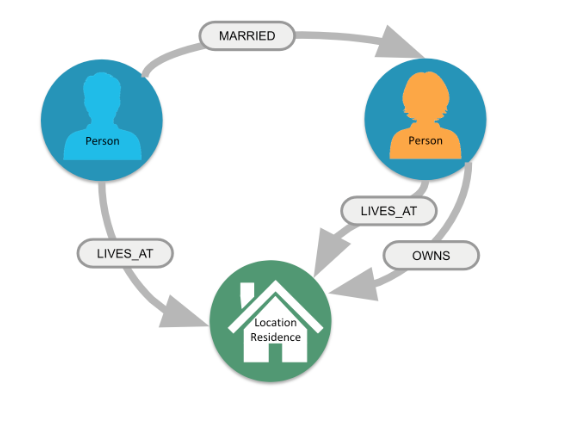
[Importing normalized data 34](#_Toc169023016)

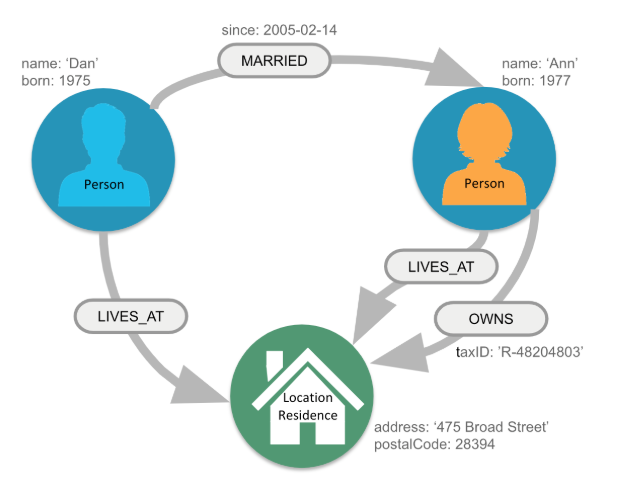
NEO4J

# Graph Database Fundamentals

A graph database can store any kind of data using a few simple concepts:

1. [](https://github.com/Wabri/LearningNeo4j/blob/master/resources/nodes.PNG)Nodes - graph data records
2. [](https://github.com/Wabri/LearningNeo4j/blob/master/resources/nodesTypes.PNG)Labels - specifies the type of the node
3. Relationships - connect nodes

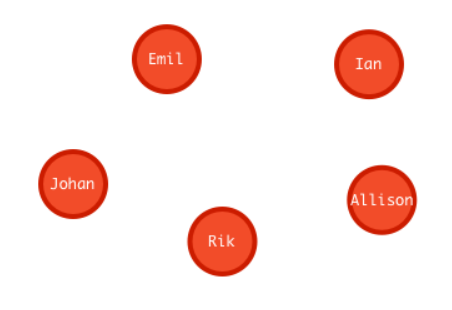
[](https://github.com/Wabri/LearningNeo4j/blob/master/resources/nodesTypesRelationships.PNG)

1. [](https://github.com/Wabri/LearningNeo4j/blob/master/resources/nodesTypesRelationshipsProperties.PNG)Properties - key-value pair properties

[](https://github.com/Wabri/LearningNeo4j/blob/master/resources/simpleGraph.PNG)The simplest graph has just a single **node** with some named values called **Properties**:

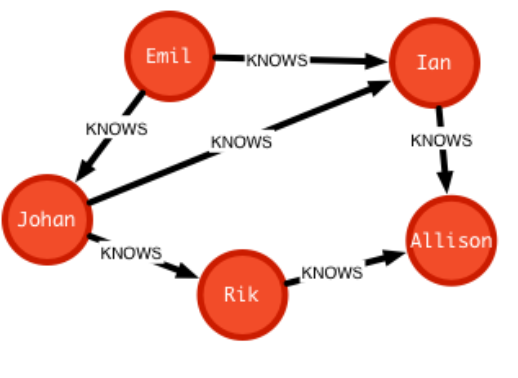
Nodes are the name for data records in a graph and the data is stored as Properties that can be simple key-value pairs.

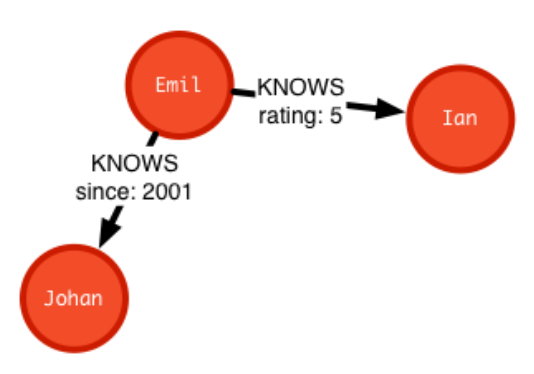
Nodes can be grouped together by applying a Label to each member. In the example above we can set to that node the label **Person**. Is important to know that a label is not an object and can't have any properties, is used only to categorize the nodes in a graph. A node can have zero or more labels based on the definition of that node.

[](https://github.com/Wabri/LearningNeo4j/blob/master/resources/moreNodes.PNG)To add more records we can simply add more nodes.

Similar nodes can have different properties with different type: string, number or even Boolean. The dimension of a graph like this can be infinite because there is no limit to the number of nodes that can be added.

One of the properties of a database is to connect data, in a graph database the link is made by **Relationships**. To associate two nodes we can add **Relationship** between them which describe how the records are related.

[](https://github.com/Wabri/LearningNeo4j/blob/master/resources/relationships.PNG)

[](https://github.com/Wabri/LearningNeo4j/blob/master/resources/relationshipProperties.PNG)A relationship are data records that need to have two properties: **direction** and **type**, and can also contains properties like nodes.

A Graph database is an online database management system with Create, Read, Update and Delete (CRUD) operations working on a graph data model. Graph database are generally build for use with [OLTP](https://github.com/Wabri/LearningNeo4j?tab=readme-ov-file#otlp) systems, they are normally optimized for transactional performance, and engineered with transactional integrity and operational availability in mind.

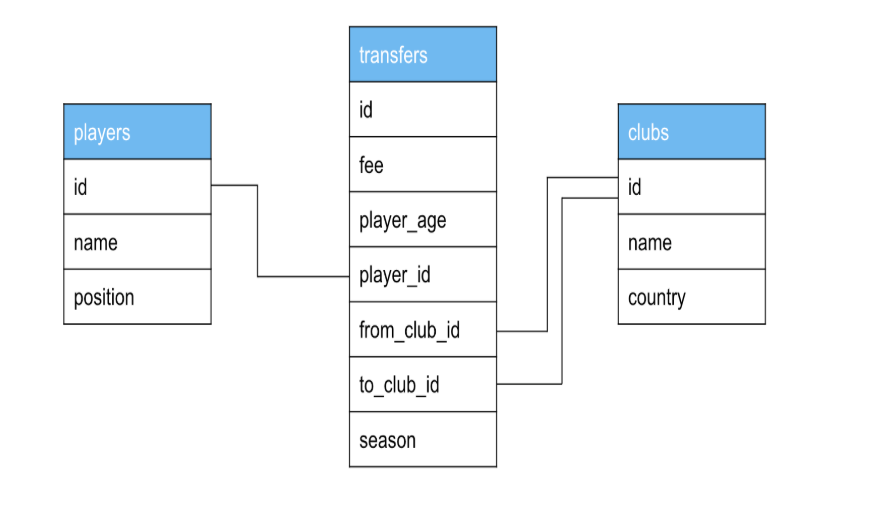
Unlike the other databases, relationships take first priority in graph databases so the foreign keys or out-of-band processing is no more necessary to link a data to another.

By assembling the simple abstractions of nodes and relationships into connected structures, graph databases enable us to build sophisticated models that map closely to our problem domain.

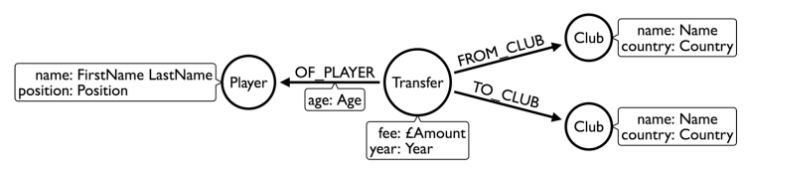
Many applications' data is modeled as relational data, indeed there are some similarities between a relational model and a graph model:

| **Relational** | **Graph** |
| --- | --- |
| Each column must have field value | Nodes with the same label aren’t' required to have the same set of properties |
| Joins are calculated at query time | Relationships are stored on disk when they are created |
| A row can belong to one table | A node can have many labels |
| Try to get the schema defined and then make minimal changes to it after that | It's common for the schema to evolve with the application |
| More abstract focus when modeling | Common to use actual data items when modeling |

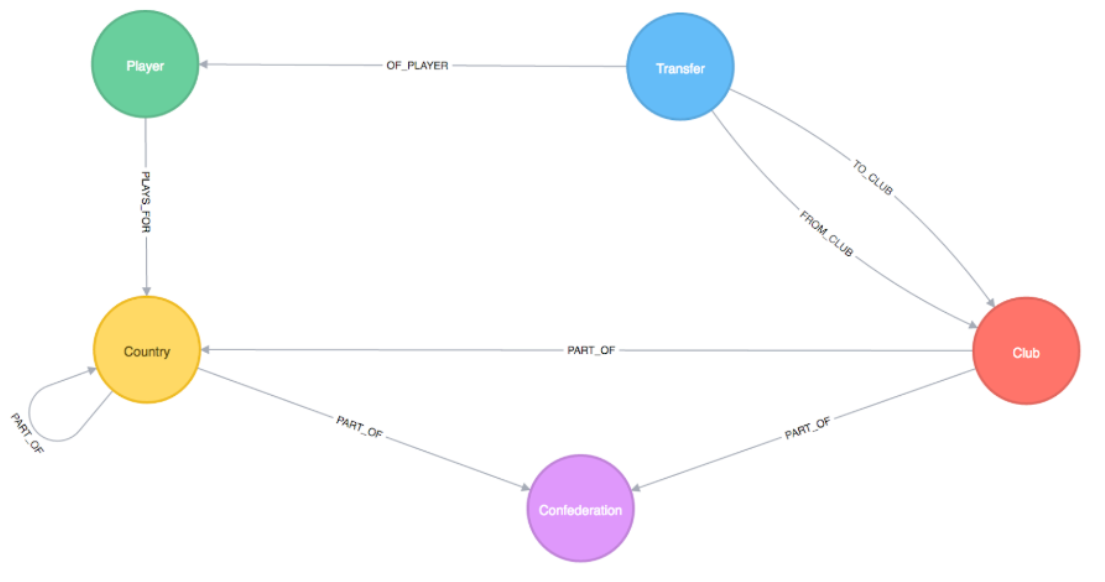
Here is the relational model:

[](https://github.com/Wabri/LearningNeo4j/blob/master/resources/relationalModelClubs.PNG)

And here is the corresponding graph model:

[](https://github.com/Wabri/LearningNeo4j/blob/master/resources/graphModelClubs.PNG)

The graph model can be more versatile and can be upgrade without efforts, for example we want to add the confederation and country:

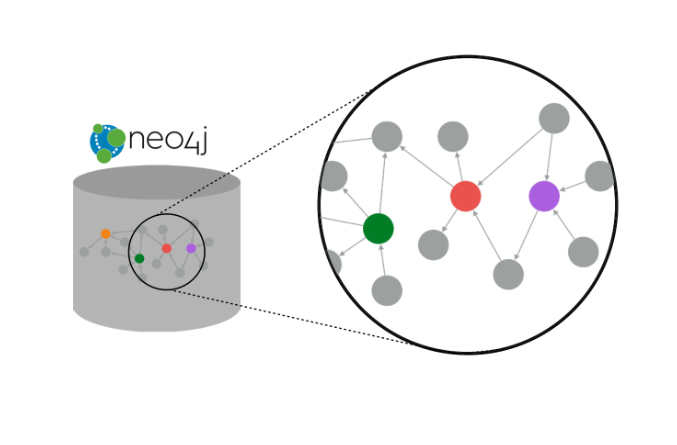
[](https://github.com/Wabri/LearningNeo4j/blob/master/resources/graphModelClubsExtend.PNG)

# NEO4J BASICS

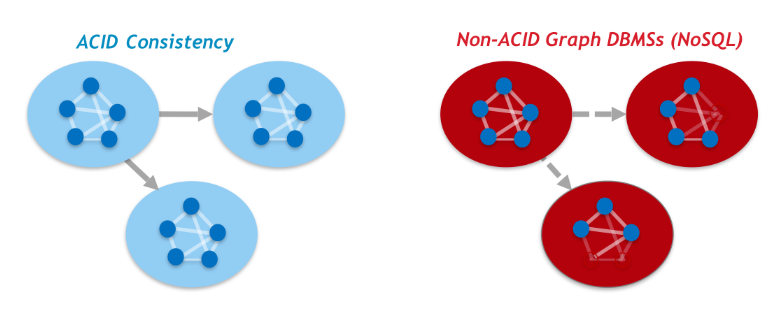
**Neo4J** is a high performance graph store with all the feature expected of a mature and robust database. The network structure is made by nodes and relationships rather than static tables.

Some definitions:

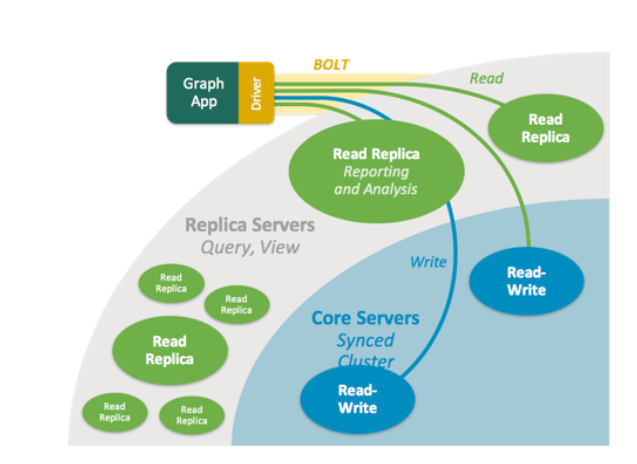
## Index free adjacency

[](https://github.com/Wabri/LearningNeo4j/blob/master/resources/neo4jIndex.PNG)With index free adjacency, when a node or relationship is written to the database, it is stored in the database as connected and any subsequent access to the data is done using pointer navigation which is very fast. Since Neo4j is a native graph database, it supports very large graphs where connected data can be traversed in constant time without the need for an index.

## ACID

[](https://github.com/Wabri/LearningNeo4j/blob/master/resources/neo4jACID.PNG)Transnationality is very important for robust applications that require an atomicity, consistency, isolation, and durability guarantees for their data. If a relationship between nodes is created, not only is the relationship created, but the nodes are updated as connected. All of these updates to the database must all succeed or fail.

## Clusters

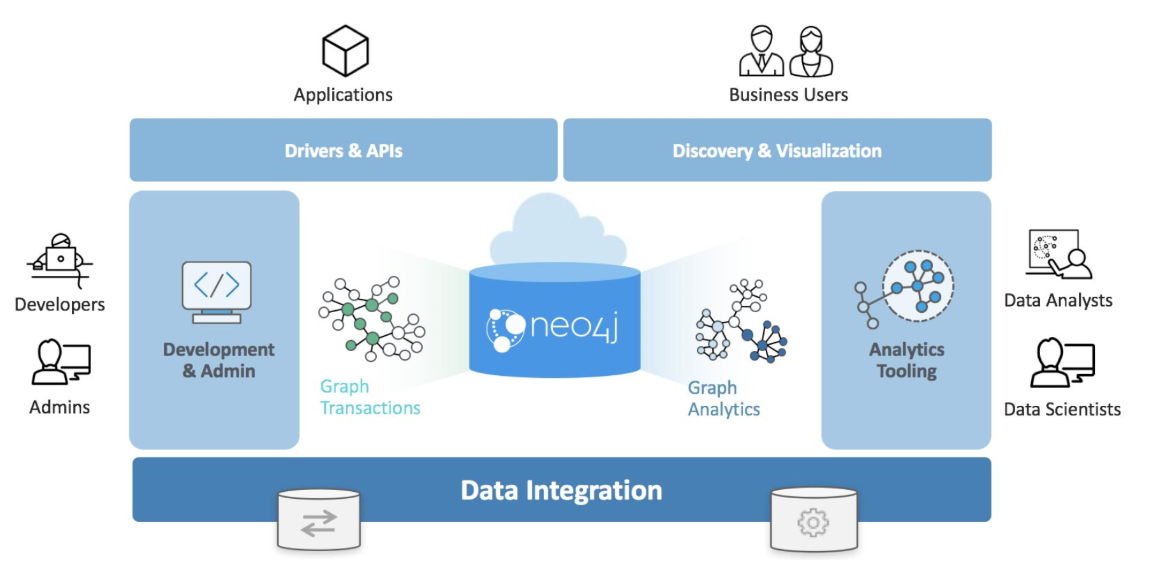
[](https://github.com/Wabri/LearningNeo4j/blob/master/resources/neo4jCluster.PNG)Neo4j supports clusters that provide high availability, scalability for read access to the data and failover which is important to many enterprises.

## Graph engine

The Neo4j graph engine is used to interpret Cypher statements and also executes kernel-level code to store and retrive data, whether it is on disk, or cached in memory.

## Bolt

Neo4j supports Java, JavaScript, Python, C#, and Go drivers that use Neo4j's bolt protocol for binary access to the database layer. Bolt is an efficient binary protocol that compresses data sent over the wire as well encrypting the data. It's possible to create a java application that uses the bolt driver to access the Neo4j database and the application may use other packages that allow data integration between Neo4j and other data stores or uses as common framework such as spring.

[](https://github.com/Wabri/LearningNeo4j/blob/master/resources/neo4jStructure.PNG)

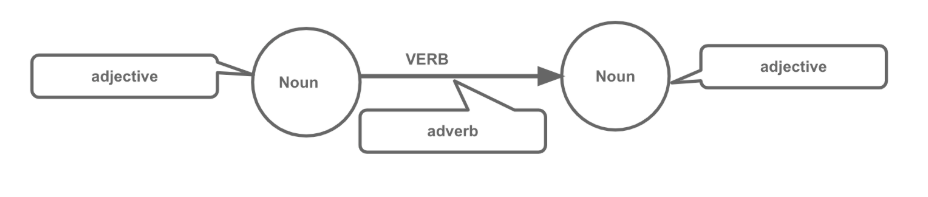
# Cypher language

Neo4J's Cypher language is purpose built for working with graph data, is a declarative query language that allows for expressive and efficient querying and updating of graph data. It uses patterns to describe graph data and is familiar to sql-like clauses. This query language allows users to store and retrive data from the Neo4J graph database with a visual and logical syntax to match patterns of nodes and relationships in the graphs. It allow to state what we want to select, insert, update, or delete from our graph data without a description of exactly how to do it:

**" Describing what to find and not how to find it "**

This means that complex database queries can easily be expressed through Cypher, allowing you to focus on your domain instead of getting lost in the syntax of database access. Also give an expressive and efficient query to handle needed create, read, update, and delete functionality (also known as CRUD operations).

The unwritten rule wants to represents the nouns as the nodes of the graph, the verbs as the relationships, the adjectives and adverbs are the properties:

[](https://github.com/Wabri/LearningNeo4j/blob/master/resources/unwrittenRule.PNG)

Graph patterns are expressed in Cypher using ASCII-art like syntax to make queries more self-explanatory:

* **NODES** uses a pair of parentheses like () or (node) to represent a node, similar to a circle on whiteboard. An anonymous node () represents one or more nodes during a query processing where there are no restrictions of the type of the node, a name inside the parentheses (node) tells the query processor that for this query is used the variable called node to represents all the nodes of the graph.
* **LABELS** are used to group nodes and filter queries against the graph and is defined with a colon (:Label). A node can have zero or more labels for example (node), (node:Label), (node:Label1:Label2), (:Label), (:Label1:Label2).
* **RELATIONSHIPS** are defined within square brackets [] and optionally we can specify type and direction like ()<-[:RELATIONSHIP]-().
* **ALIASES** are used to referred elements to later in the query defined by a name before a name like (node1:Label1)<-[relationship:RELATIONSHIP]-(node2:Label2) where node1, node2 and relationship are aliases.
* **Predicates** are filters that can be applied to limit the matching paths: Boolean logic operators, regular expressions and string comparison operators.

## Match

The most widely used Cypher clause is **MATCH**, this performs a pattern match against the data in the graph. During the query processing, the graph engine traverses the graph to find all nodes that match the graph pattern.

A query with match need to be present with the **RETURN** clause. This clause must be the last of a query to the graph. Here some examples:

// returns all nodes in the graph

MATCH (variable)

RETURN variable

// returns all Label nodes in the graph

MATCH (variable:Label)

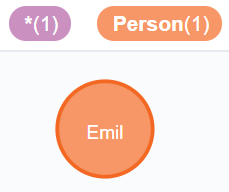
RETURN variable // returns

When you specify a pattern for a **MATCH** clause, you should always specify a node label if possible. In doing so, the graph engine uses an index to retrive the nodes which will perform better than not using a label for the **MATCH**.

**Type of query output**

The output of a query can be different:

* by **graph**:

[](https://github.com/Wabri/LearningNeo4j/blob/master/resources/matchEmilReturnG.PNG)

* by **table**:
* {
* "name": "Emil",
* "from": "Sweden",
* "klout": 99

}

* by **text**:

Explain

╒══════════════════════════════════════════╕

│"ee" │

╞══════════════════════════════════════════╡

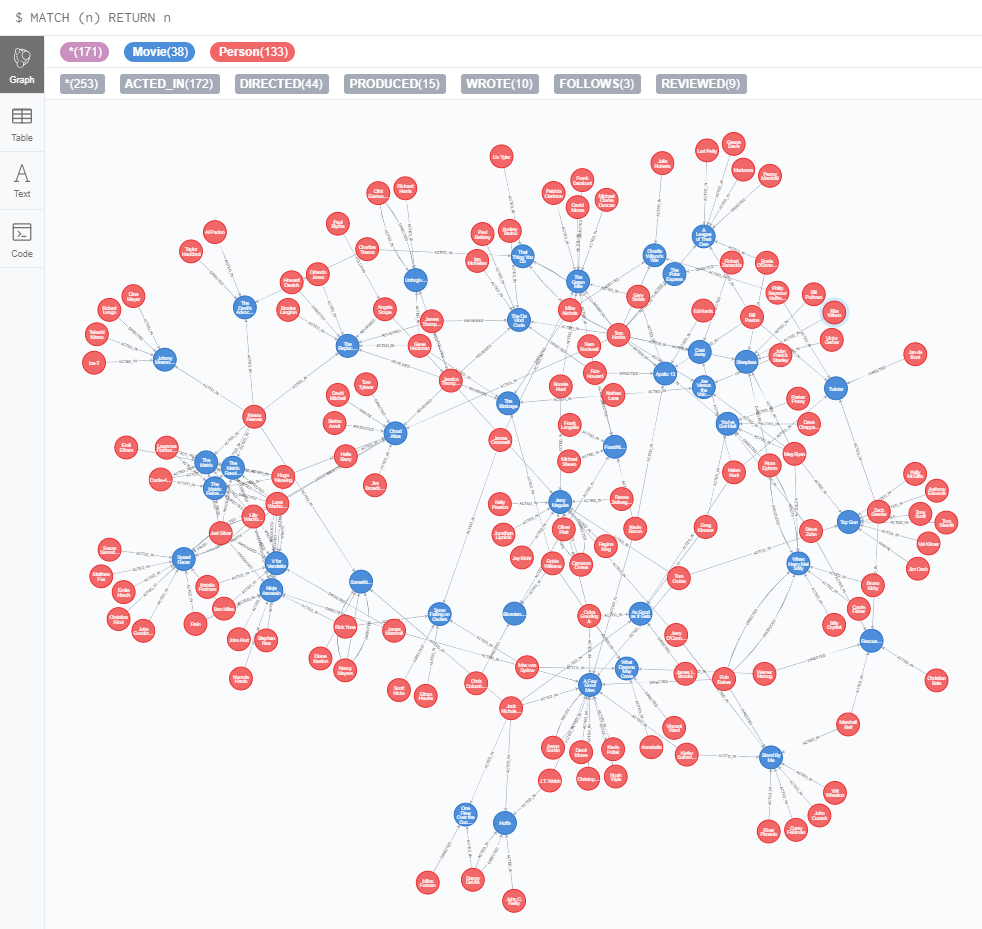
│{"name":"Emil","from":"Sweden","klout":99}│

└──────────────────────────────────────────┘

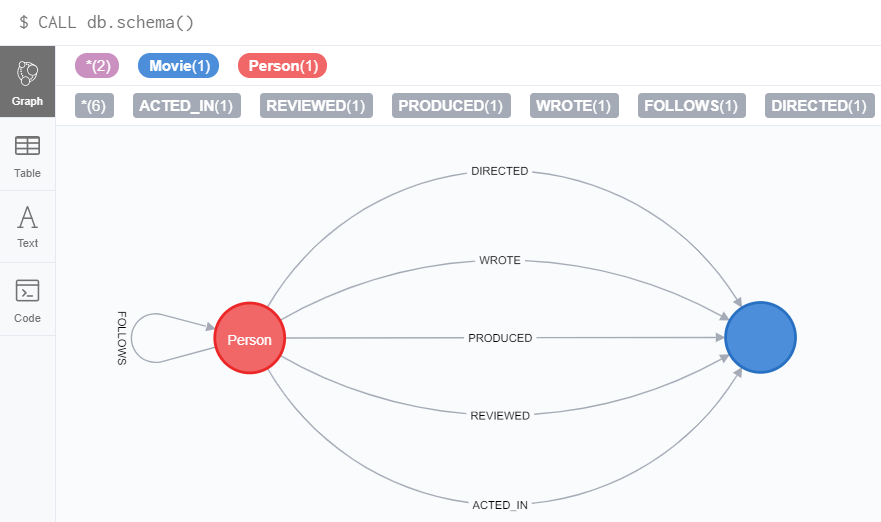
EXAMPLES:

1. Retrive all nodes from the database

MATCH (n)

[](https://github.com/Wabri/LearningNeo4j/blob/master/resources/partOneExercise_1_1.PNG)RETURN n

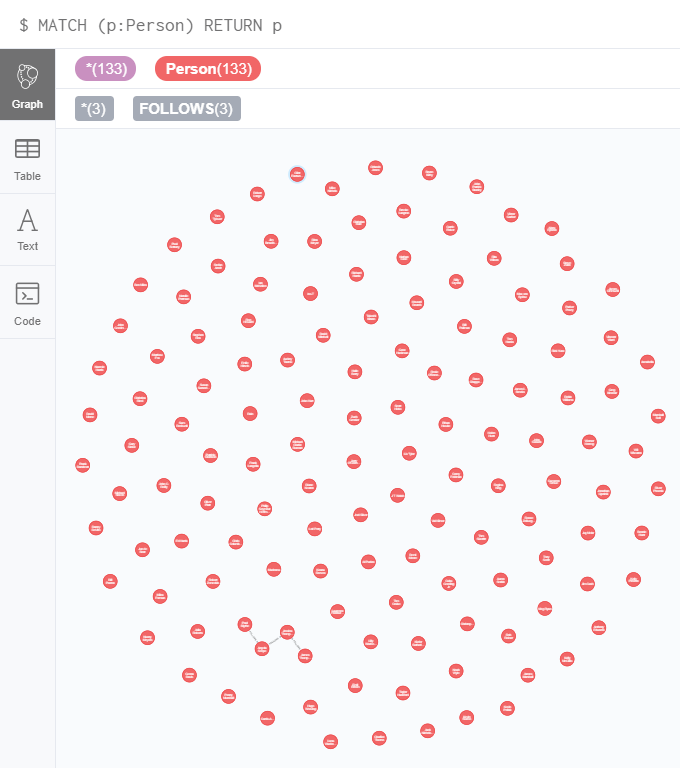
1. Examine the schema of your database

[](https://github.com/Wabri/LearningNeo4j/blob/master/resources/partOneExercise_1_2.PNG)CALL db.schema()

1. Retrive all Person nodes

MATCH (p:Person)

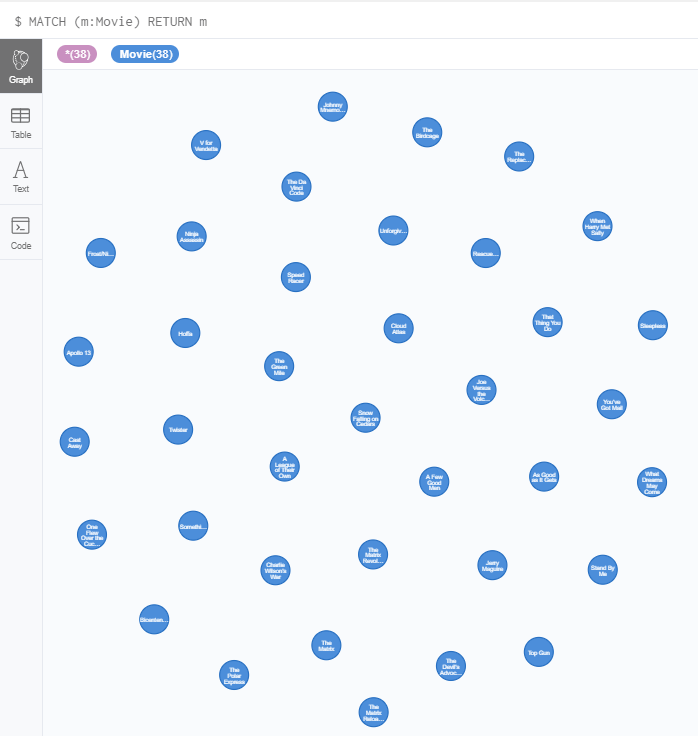
RETURN p

[](https://github.com/Wabri/LearningNeo4j/blob/master/resources/partOneExercise_1_3.PNG)

1. Retrive all Movie nodes

MATCH (m:Movie)

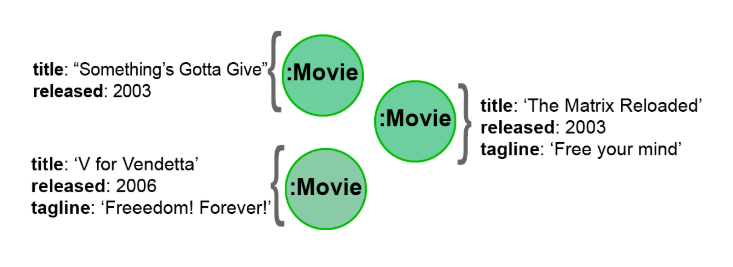
RETURN m

[](https://github.com/Wabri/LearningNeo4j/blob/master/resources/partOneExercise_1_4.PNG)

## Properties

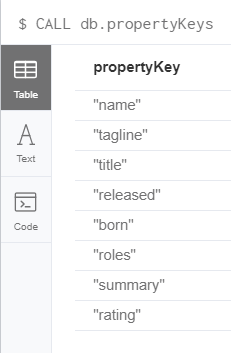
In Neo4j a node can have properties that are used for further define a node. A property is identified with a key and defined for a node and not for a type of node. All nodes of the same type need not have the same properties.

For example in the Movie graph all Movie nodes have both title and released properties, however it is not requirement that every Movie node has a property tagline:

[](https://github.com/Wabri/LearningNeo4j/blob/master/resources/taglineMovie.PNG)

Properties can be used to filter queries so that a subset of the graph is retrieved. In addition, with the RETURN clause, you can return property values from the retrieved nodes, rather than the nodes.

The property keys of a graph can be view by execute CALL db.propertyKeys which call the Neo4j library method that returns the property keys for the graph. For example run this command in the movie graph returns the result stream contains all property keys in the graph:

[](https://github.com/Wabri/LearningNeo4j/blob/master/resources/propertyKeys.PNG)

EXAMPLE:

* 1. Retrieve all Movie nodes that have a released property value of 2003.

MATCH (m:Movie {released: 2003})

RETURN m

* 1. View the retrieved results as a table.

MATCH (m:Movie {released: 2003})

RETURN m

* 1. : Query the database for all property keys.

CALL db.propertyKeys

* 1. Retrieve all Movies released in a specific year, returning their titles.

MATCH (m:Movie {released: 2006})

RETURN m.title

* 1. Display title, released, and tagline values for every Movie node in the graph.

MATCH (m:Movie)

RETURN m.title, m.released, m.tagline

* 1. Display more user-friendly headers in the table.

MATCH (m:Movie)

RETURN m.title AS `Movie title`, m.released AS `Released date`, m.tagline AS `Tag line`.

## Relationships

A relationship is a directed connection between two nodes that has a relationship type (name). In addition, a relationship can have properties, just like nodes. In a match clause it's possible to specify nodes and their relationships to traverse the graph and quickly find the data of interest:

* () // a node
* ()--() // 2 nodes have some type of relationship
* ()-->() // the first node has a relationship to the second node
* ()<--() // the second node has a relationship to the first node The relationship can be specified with or without direction.

Here some examples for retrieving a set of nodes that satisfy one or more directed and type relationships:

MATCH (node1)-[:REL\_TYPE]->(node2)

RETURN node1, node2

MATCH (node1)-[:REL\_TYPEA | :REL\_TYPEB]->(node2)

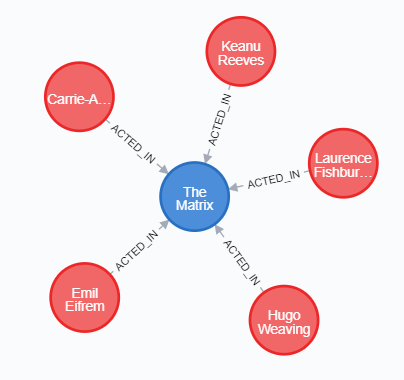
RETURN node1, node2

Where:

|  |  |
| --- | --- |
| node1, node2 | is a specification of a node where you may include node labels and property values for filtering |
| :REL\_TYPE | is the type for the relationship from node1 to node2 |
| :REL\_TYPEA, :REL\_TYPEB | are the relationships from node1 to node2, the nodes are returned if at least one of the relationships exists |

In the movie graph to retrive the nodes Person that have acted in the Movie "The Matrix" we need to use relationships:

MATCH (node1:Person)-[relation:ACTED\_IN]->(node2:Movie {title: "The Matrix"})

[](https://github.com/Wabri/LearningNeo4j/blob/master/resources/relationshipActorMatrix.PNG)RETURN node1, relation, node2

There is a build-in function type() that returns the relationship type of a relationship. Here is an example where we use the rel variable to hold the relationships retrieved:

MATCH (p:Person)-[rel]->(:Movie {title: "The Matrix"})

RETURN p.name, type(rel)

The output list will be:

| **p.name** | **type(rel)** |
| --- | --- |
| "Hugo Weaving" | "ACTED\_IN" |
| "Lilly Wachowski" | "DIRECTED" |
| "Laurence Fishburne" | "ACTED\_IN" |
| "Lana Wachowski" | "DIRECTED" |
| "Keanu Reeves" | "ACTED\_IN" |
| "Carrie-Anne Moss" | "ACTED\_IN" |
| "Joel Silver" | "PRODUCED" |
| "Emil Eifrem" | "ACTED\_IN" |

Even relationships can have properties, this enables the graph model to provide more data about the relationships between the nodes. Just as can be specify property values for filtering nodes for a query, you can specify property values for a relationship. Here is an example with the movie graph:

// Returns the name of the person who gave the movie "the da Vinci code" a rating of 65

MATCH (p:Person)-[:REVIEWED {rating: 65}]->(:Movie {title: "The Da Vinci Code"})

RETURN p.name

Since relationships are directional queries can have multiple type of matching:

* right direction
* MATCH (n)-[r]->(m)

RETURN n, r, m

* left direction
* MATCH (n)<-[r]-(m)

RETURN n, r, m

* both direction
* MATCH (n)-[r]-(m)

RETURN n, r, m

* traversing relationships
* MATCH (n)-[r]->(m)-[p]->(o)

RETURN n, m, o

with query like this it can be useful to assign a variable to the path and return the path:

MATCH path = (n)-[r]->(m)-[p]->(o)

RETURN path

* Centralizing relationships
* MATCH (n)-[r]->(m)<-[p]-(o)

RETURN n, m, o

**Style recommendations**

* Node labels are CamelCase and begin with an upper-case letter, like Person or Network Address.
* Property keys, variables, parameters, aliases, and functions are camelCase and begin with a lower-case letter, like title or business Address.
* Relationship type are in upper-case and can use the underscore, like ACTED\_IN or FOLLOWS.
* Cypher keywords are upper-case, like MATCH or RETURN.
* String constants are in single quotes, unless the string contains a quote or apostrophe, like 'The Matrix' or "Something's Got to Give".
* Specify variables only when needed for use later in the cypher statement.
* Place named nodes and relationships before anonymous nodes and relationships in the MATCH clauses when possible.
* Specify anonymous relationships with -->, --, or <--.

**EXAMPLE:**

1. Display the schema of the database.

CALL db.schema

1. Retrieve all people who wrote the movie Speed Racer.

MATCH (p:Person)-[:WROTE]->(:Movie {title: "Speed Racer"})

RETURN p

1. Retrieve all movies that are connected to the person, Tom Hanks.

MATCH (m:Movie)--(:Person {name: 'Tom Hanks'})

RETURN m

1. Retrieve information about the relationships Tom Hanks had with the set of movies retrieved earlier.

MATCH (m:Movie)-[relation]-(:Person {name: 'Tom Hanks'})

RETURN m.title, type(relation)

1. Retrieve information about the roles that Tom Hanks acted in

MATCH (m:Movie)-[acted:ACTED\_IN]-(:Person {name: 'Tom Hanks'})

RETURN m.title, acted.roles

## Where

The most common clause to filter queries is WHERE that follows a MATCH clause. This clause is the answer for "how we filter the result for a particular match", so this filter all of the nodes and relationships. In the WHERE clause it is possible to place conditions that are evaluated at runtime to filter the query. The potential of this clause is that is possible to specify complex conditions for the query.

Some examples:

1. Example 1
2. MATCH (m:Movie)
3. WHERE m.title = "The Matrix"

RETURN m

1. Example 2
2. MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)
3. WHERE m.released = 2008

RETURN p, m

1. Example 3
2. MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)
3. WHERE m.released = 2008 OR m.released = 2009

RETURN p, m

This clause accept conditions that return a Boolean value of true or false. It can be use several comparison operators: **=**, **<>**, **<**, **>**, **<=**, **>=**, **IS NULL**, **IS NOT NULL**, **=~**. There are 4 Boolean operators that it can use: **AND**, **OR**, **XOR**, **NOT**.

An example:

MATCH (p:Person)-[r:ACTED\_IN]->(m:Movie)

WHERE

m.released > 2000 OR

(1994 < m.released <= 1997 AND m.title='As Good as It Gets')

RETURN p.name, m.title, m.released

It is possible to filter node labels in the WHERE clause, for example this two queries:

MATCH (p:Person)

RETURN p.name

MATCH (p:Person)-[:ACTED\_IN]->(:Movie {title: 'The Matrix'})

RETURN p.name

can be rewritten using WHERE clauses:

MATCH (p)

WHERE p:Person

RETURN p.name

MATCH (p)-[:ACTED\_IN]->(m)

WHERE p:Person AND m:Movie AND m.title = 'The Matrix'

RETURN p.name

Since we are talking about graph database not all the nodes with the same label have the same properties, with the WHERE clause and the build-in function exists(property) we can filter the nodes that doesn't have value for the property requested. For example:

MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WHERE p.name='Jack Nicholson' AND exists(m.tagline)

RETURN m.title, m.tagline

There are also a set of string-related keywords to test string property values: STARTS WITH, ENDS WITH, and CONTAINS.

MATCH (p:Person)

WHERE p.name =~'Tom.\*'

RETURN p.name

The query above retrive all Person nodes with a name property that begins with Tom, the result can be something like this:

| **p.name** |
| --- |
| "Tom Cruise" |
| "Tom Hanks" |
| "Tom Skerritt" |
| "Tom Tykwer" |

Some more filtering for relationships can be used during a query, for example can be possible to use the NOT in a WHERE clause:

MATCH (p:Person)-[:WROTE]->(m:Movie)

WHERE NOT exists( (p)-[:DIRECTED]->() )

RETURN p.name, m.title

This query exclude Person who directed and not wrote the movie.

One more filtering is IN that can be used to compare each property with values on the list:

MATCH (p:Person)

WHERE p.born IN [1965, 1970]

RETURN p.name as name, p.born as yearBorn

and the return is:

| **name** | **yearBorn** |
| --- | --- |
| "Lana Wachowsky" | 1965 |
| "Jay Mohr" | 1970 |
| "River Phoenix" | 1970 |
| "Brooke Langton" | 1970 |

**EXAMPLE:**

* 1. Retrieve all movies that Tom Cruise acted in.

MATCH (tom:Person {name: 'Tom Cruise'})-[:ACTED\_IN]-(movie:Movie)

RETURN movie.title

or

MATCH (tom:Person)-[:ACTED\_IN]-(movie:Movie)

WHERE tom.name = 'Tom Cruise'

RETURN movie.title

* 1. Retrieve all actors that were born in the 70’s, return name and year born.

MATCH (p:Person)-[:ACTED\_IN]->(:Movie)

WHERE 1970 <= p.born <= 1979

RETURN p.name AS Name, p.born AS `Year Born`

* 1. Retrieve the actors who acted in the movie The Matrix who were born after 1960, return name and year born.

MATCH (act:Person)-[:ACTED\_IN]->(m:Movie)

WHERE m.title = 'The Matrix' AND act.born > 1960

RETURN act.name as Name, act.born as `Year Born`

* 1. Retrieve all movies released in 2000 by testing the node label and released property, return the title of the movie.

MATCH (mov:Movie)

WHERE mov.released = 2000

RETURN mov.title

* 1. Retrieve all people that wrote movies by testing the relationship between two nodes, return the name of the people and the title of the movie.

MATCH (p)-[rel]->(mov)

WHERE p:Person AND type(rel) ='WROTE' AND mov:Movie

RETURN p.name AS Name, mov.title AS `Movie title`

* 1. Retrieve all people in the graph that do not have the born property and return their name.

MATCH (p)

WHERE p:Person AND NOT exists(p.born)

RETURN p.name AS Name

## Aggregation in Cypher

In Cypher is not need to specify a grouping key, all non-aggregated result columns become grouping keys, The grouping is implicitly done based upon the fields in the RETURN clause.

For example, in Cypher statement, all rows returned with the same values for a.name and d.name are counted and only return once:

MATCH (a)-[:ACTED\_IN]->(m)<-[:DIRECTED]-(d)

RETURN a.name, d.name, count(\*)

| **a.name** | **d.name** | **count(\*)** |
| --- | --- | --- |
| "Emil Eifrem" | "Lana Wachowski" | 1 |
| "Hugo Weaving" | "Lana Wachowski" | 4 |
| "Laurence Fishburne" | "Lana Wachowski" | 3 |
| "Carrie-Anne Moss" | "Lana Wachowski" | 3 |
| "Keanu Reeves" | "Lana Wachowski" | 3 |
| "Emil Eifrem" | "Lilly Wachowski" | 1 |
| "Hugo Weaving" | "Lilly Wachowski" | 4 |
| "Laurence Fishburne" | "Lilly Wachowski" | 3 |
| "Carrie-Anne Moss" | "Lilly Wachowski" | 3 |
| "Keanu Reeves" | "Lilly Wachowski" | 3 |
| "Al Pacino" | "Taylor Hackford" | 1 |

This function is very useful when you want to count the number of occurrences of a particular query result. It's possible to specify the occurrences of an alias count(n) and the graph engine calculates the number of occurrences of n. If we want to calculates the number of rows retrieved, including those with null values the count argument need to be a \*. Last one is the count() without argument and this will implicit group by based upon the aggregation.

There are more aggregating functions such as min() or max() that can also use in queries.

1. **Collecting results**

Cypher has a built-in function collect() that enables you to aggregate value into a list:

MATCH (p:Person)-[:ACTED\_IN]->(m:Movie)

WHERE p.name = 'Tom Cruise'

RETURN collect(m.title) AS `movies for Tom Cruise`

And the result will be a list called **movies for Tom Cruise** with the values ["Jerry Maguire", "Top Gun", "A Few Good Men"].

1. **Additional processing using WITH**

During the execution of a MATCH clause, is possible to specify some intermediate calculations or values that will be used for further processing of the query, or for limiting the number of results before further processing is done. With the WITH clause it's possible to perform intermediate processing or data flow operations.

Here is an example:

MATCH (p:Person)

WITH p, size((p)-[:ACTED\_IN]->(:Movie)) as movies

WHERE movies>=5

OPTIONAL MATCH (p)-[:DIRECTED]->(m:Movie)

RETURN p.name, m.title

This is a simple query to retrieve all the actor that are acted in at least 5 movies and if they also directed a movie than return the name of that movie.

| **p.name** | **m.title** |
| --- | --- |
| "Keanu Reeves" | null |
| "Hugo Weaving" | null |
| "Jack Nicholson" | null |
| "Meg Ryan" | null |
| "Tom Hanks" | "That Thing You Do" |

**EXAMPLE:**

1. Write a Cypher query that retrieves all movies that Gene Hackman has acted it, along with the directors of the movies. In addition, retrieve the actors that acted in the same movies as Gene Hackman. Return the name of the movie, the name of the director, and the names of actors that worked with Gene Hackman.

MATCH (gene:Person)-[:ACTED\_IN]->(movie:Movie)

WHERE gene.name = 'Gene Hackman'

OPTIONAL MATCH

(other:Person)-[:ACTED\_IN]->(movie),

(dir:Person)-[:DIRECTED]->(movie)

WITH

movie,

collect(other.name) AS Actors,

collect(dir.name) AS Directors

RETURN

movie.title AS `Title of movie`,

Actors AS `Co-Actors`,

Directors

1. : Retrieve all nodes that the person named James Thompson directly has the FOLLOWS relationship in either direction.

MATCH (james:Person)-[:FOLLOWS]-(other:Person)

WHERE james.name = 'James Thompson'

RETURN james, other

1. Modify the query to retrieve nodes that are exactly three hops away.

MATCH (james:Person)-[:FOLLOWS\*3]-(other:Person)

WHERE james.name = 'James Thompson'

RETURN james, other

1. Modify the query to retrieve nodes that are one and two hops away.

MATCH (james:Person)-[:FOLLOWS\*1..2]-(other:Person)

WHERE james.name = 'James Thompson'

RETURN james, other

1. : Write a Cypher query to retrieve all people in the graph whose name begins with Tom and optionally retrieve all people named Tom who directed a movie.

MATCH (tom:Person)

WHERE toLower(tom.name) STARTS WITH 'tom'

OPTIONAL MATCH (tom)-[:DIRECTED]->(movie:Movie)

WITH tom, collect(movie.title) as DirMov

RETURN tom.name AS `Tom\* name`, DirMov AS `Directed movies`

1. Retrieve actors and the movies they have acted in, returning each actor’s name and the list of movies they acted in.

MATCH (actor:Person)-[:ACTED\_IN]->(movie:Movie)

WITH actor.name AS Actor, collect(movie.title) as Movies

RETURN Actor, Movies

1. Retrieve the movies that have at least 2 directors, and optionally the names of people who reviewed the movies.

MATCH (dir:Person)-[:DIRECTED]->(movie:Movie)

WITH movie, count(dir) AS numDir

WHERE numDir >= 2

OPTIONAL MATCH (movie)<-[:REVIEWED]-(reviewer:Person)

WITH movie.title AS title, collect(reviewer.name) as listRev

RETURN title, listRev AS `reviewer`

## Create

Recall that a node is an element of a graph representing a domain entity that has zero or more labels, properties, and relationships to or from other nodes in the graph.

When you create a node, you can add it to the graph without connecting it to another node.

Here is the simplified syntax for creating a node:

CREATE (optionalVariable optionalLabels {optionalProperties})

Let's create a small social graph using this query language.

If you plan on referencing the newly created node, you must provide a variable. Whether you provide labels or properties at node creation time is optional. In most cases, you will want to provide some label and property values for a node when created. This will enable you to later retrive the node. Provided you have a reference to the node.

Here some examples:

* 1 -> Create node of type *Movie* with the *title Batman Begins*:

CREATE (:Movie {title: 'Batman Begins'})

* 2 -> Create a node with two labels to the graph of types *Movie* and *Action* with the *title Batman Begins*:

CREATE (:Movie:Action {title: 'Batman Begins'})

* 3 -> Create the same node of example 2 and return the title:
* CREATE (m:Movie:Action {title: 'Batman Begins'})

RETURN m.title

* 4 -> Create the same node but with more properties:

CREATE (:Movie:Action {title: 'Batman Begins', released: 2005})

When the graph engine creates a node, it automatically assigns a read-only, unique ID to the node but this value is not accessible like the other properties but is an internal value.

If all of these 4 queries are executed, this will generate 4 different nodes this is because the graph engine will create a node with the same properties of a node that already exists. To prevent this "*issue*" you can use the **MERGE** clause rather than **CREATE** or you can add constraints to the graph.

**More create at once**

**Create** clauses can create many nodes at once:

CREATE

(:Person {name: 'Michael Caine', born: 1933}),

(:Person {name: 'Liam Neeson', born: 1952})

CREATE

(:Person {name: 'Katie Holmes', born: 1978}),

(:Person {name: 'Benjamin Melniker', born: 1913})

There are two ways to create multiple nodes at once: by separating the nodes specified with commas, or by specifying multiple CREATE statements.

**Adding properties to a node**

The **SET** clause can be use even to add properties to a node.

Some examples:

SET x.propertyName = value

SET x.propertyName1 = value1, x.propertyName2 = value2

SET x = {propertyName1: value1, propertyName2: value2}

SET x += {propertyName1: value1, propertyName2: value2}

If the property does not exist, it is added to the node. If the property exists, its value is updated. If the value specified is null, the property is removed.

The type of data for a property is not enforced, so you can assign a string value to a property that was once a numeric value and vice versa.

When the assignment used is the = than the properties and their values must exists and their values are overwritten; however, += is used to update if exists already, alternatively the properties and values are created.

Example: Add the properties *released* and *lengthInMinutes* to the movie *Batman Begins*.

MATCH (m:Movie)

WHERE m.title = 'Batman Begins'

SET m.released = 2005, m.lengthInMinutes = 140

RETURN m

MATCH (m:Movie)

WHERE m.title = 'Batman Begins'

SET m = {title: 'Batman Begins',

released: 2005,

lengthInMinutes: 140,

videoFormat: 'DVD',

grossMillions: 206.5}

RETURN m

MATCH (m:Movie)

WHERE m.title = 'Batman Begins'

SET m += {

grossMillions: 300,

awards: 66

}

RETURN m

1. **Removing properties from a node**

There are two ways that you can remove a property from a node:

1. Using the **REMOVE** keyword:

REMOVE x.propertyName

1. Set the property's value to null

SET x.propertyName = null

For example if we want to remove the video format and the gross millions from the batman begins film we can execute this query:

MATCH (m:Movie)

WHERE m.title = 'Batman Begins'

SET m.grossMillions = null

REMOVE m.videoFormat

RETURN m

## Deleting nodes and relationships

If a node has no relationships to any other nodes, it's possible to simply delete it from the graph using the **DELETE**.

The graph engine return errors if attempt to delete a node in the graph that has relationships in or out of the node, this is because deleting such a node will leave *orphaned* relationships in the graph.

To delete a relationship between nodes it's necessary to find it first in the graph and then deleting it.

Example:

* Delete the ACTED\_IN relationship between Christian Bale and the movie Batman Begins:
* MATCH (a:Person)-[rel:ACTED\_IN]->(m:Movie)
* WHERE a.name = 'Christian Bale' AND 'm.title = 'Batman Begins'
* DELETE rel

RETURN a, m

* Delete the node *Benjamin Melniker* and his relationships to movie nodes. To make this works it's necessary to remove the relationships to prevent errors:
* MATCH (p:Person)-[rel:PRODUCED]->(:Movie)
* WHERE p.name = 'Benjamin Melniker'

DELETE rel, p

The most efficient way to delete a node and its corresponding relationships is to specify **DETACH DELETE**. With this specification, the relationships to and from the node are deleted then the node is deleted.

Examples:

* This gives errors:
* MATCH (p:Person)
* WHERE p.name = 'Liam Neeson'

DELETE p

* This not gives errors:
* MATCH (p:Person)
* WHERE p.name = 'Liam Neeson'

DETACH DELETE p

## Merge

The **MERGE** clause is used either to create new nodes and relationships or to make structural changes to existing nodes and relationships.

Recap of the CREATE clause:

|  | **The result with CREATE** |
| --- | --- |
| Node | If a node with the same property value exists, a duplicate node is created |
| Label | If the label already exists for the node, the node is not updated |
| Property | If the node or relationship property already exists, it is updated with the new value |
| Relationship | If a relationship exists, a duplicate relationship is created |

It's difficult to have a graph where we want some duplicates, and the solution for this is the MERGE clause. This clause is used to find elements in the graph and if the element is not found then it will be created:

* Create a unique node based on label and key information for a property and if it exists, optionally update it
* Create a unique relationship
* Create a node and relationship to it uniquely in the context of another node

MERGE (variable:Label{nodeProperties})

RETURN variable

This is an example:

MERGE (a:Person {name: 'Michael Caine'}) SET a.born = 1933 RETURN a

We can repeat this query with the MERGE clause but no more Actor will be created.

This clause can be used also to create relationships:

MERGE (variable:Label {nodeProperties})-[:REL\_TYPE]->(otherNode)

RETURN variable

It's possible to leave out the direction of the relationship, if the relationship not exists than will be created assuming the left-to-right arrow.

Using this clause to create relationships is expensive and the suggestion is to do only when it's needed to ensure that a relationship is unique and you are not sure it already exists.

To prevent this the MATCH clause:

MATCH (p:Person), (m:Movie)

WHERE m.title = 'Batman Begins' AND p.name ENDS WITH 'Caine'

MERGE (p)-[:ACTED\_IN]->(m)

RETURN p,m

You must be aware of the behavior of the **MERGE** clause and how it will automatically create nodes and relationships. This clause tries to find a full pattern and if it doesn't find it, it creates that full pattern. That's why in most cases it's suggested to first **MERGE** the nodes and then the relationships afterwards.

Another property of the **MERGE** clause is **ON CREATE** used to assign specific values to a node being created as a result of an attempt to merge.

Example:

MERGE (a:Person {name: 'Sir Michael Caine'})

ON CREATE SET a.birthPlace = 'London',

a.born = 1934

RETURN a

The SET clause will be used only if the node is not found, the clause will be skipped otherwise.

Like ON CREATE it exists another clause for the matching case **ON MATCH**, used when the node/relationship already exists.

Example:

MERGE (a:Person {name: 'Sir Michael Caine'})

ON CREATE SET a.born = 1934

a.birthPlace = 'UK'

ON MATCH SET a.birthPlace = 'UK'

RETURN a

## Managing indexes

The uniqueness and node key constraints are essentially single-property and composite indexes respectively. Indexes are used to improve initial node lookup performance, but they require additional storage in the graph to maintain and also add to the cost of creating or modifying property values that are indexed. Indexes store redundant data that points to nodes with the specific property value or values. Unlike SQL, there is no such thing as a primary key in Neo4j, but nodes can have multiple properties that must be unique.

This are single-property indexes used:

* Equality checks: =
* Range comparisons: >,>=,<, <=
* List membership: IN
* String comparisons: STARTS WITH, ENDS WITH, CONTAINS
* Existence checks: exists()
* Spatial distance searches: distance()
* Spatial bounding searches: point()

Composite indexes are used only for equality checks and list membership.

Because index maintenance incurs additional overhead when nodes are created so it's not recommended to create indexes in a small graph. The indexes of a graph can be view by use the build-in command schema.

The index for a property of a node can greatly reduce the number of nodes that the engine needs to visit in order to satisfy a query.

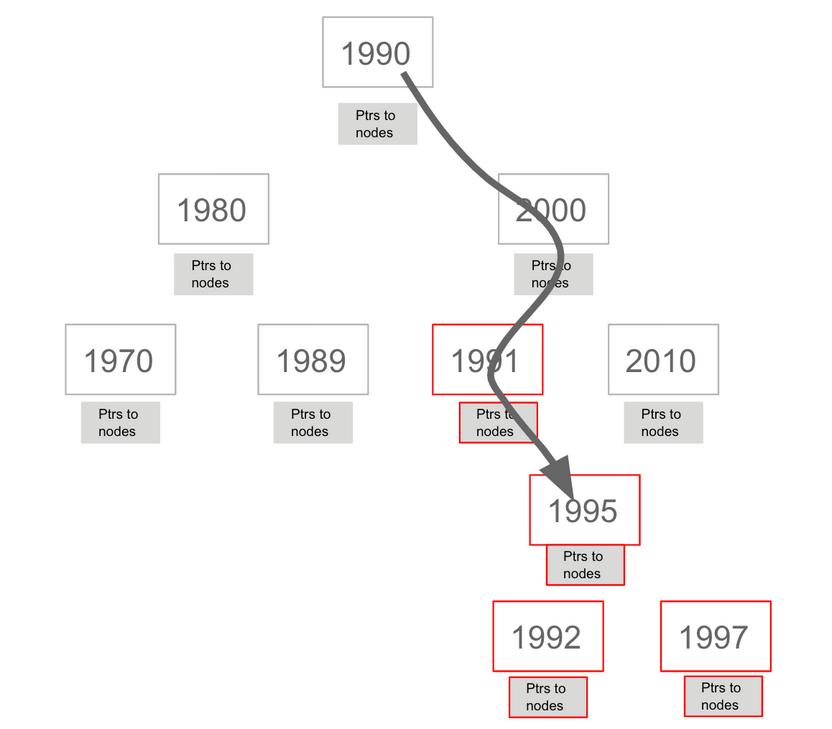
Example:

MATCH (m:Movie)

WHERE 1990 < m.released < 2000

SET m.videoFormat = 'DVD'

The graph engine will find the pointers to all nodes that satisfy the query without having to visit all of the nodes:

[](https://github.com/Wabri/LearningNeo4j/blob/master/resources/indexesRange.png)

Indexes improve graph engine performance. A unique constraint on a property is an index so it's not needed to create an index for any properties created with uniqueness constraints for, but an index on its own does not guarantee uniqueness.

To create a single-property index it's necessary to use the CREATE INDEX ON:

CREATE INDEX ON :Movie(released)

If a set of properties for a node must be unique for every node, then it's recommended to create a constraint as a node key, rather than an index. If there can be duplication for a set of property values, but the faster access it's necessary than the composite index is necessary.

Example:

MATCH (m:Movie)

WHERE m.released >= 2000

SET m.videoFormat = 'DVD';

MATCH (m:Movie)

WHERE m.released < 2000

SET m.videoFormat = 'VHS';

CREATE INDEX ON :Movie(released,videoFormat)

This query set two new properties and with the last query create composite index on these properties.

To retrive indexes there is a build-in method call :schema, similar to CALL db.constraints(), that return a list of the indexes, and the unique constraints and node key shown as indexes in the graph.

To drop an existing index created it's possible to use the clause DROP instead of CREATE:

DROP INDEX ON :Movie(released, videoFormat)

## Importing normalized data

The LOAD CSV clause parses a local in the import directory of the neo4j installation or a remote file into a stream of rows which represent maps (with Readers) or lists. Once done this it's possible to use Cypher operations to create nodes or relationships or merge existing graph.

Syntax of LOAD CSV:

LOAD CSV WITH HEADERS FROM url-value

AS row

* The **row** is a variable that is used to extract data from file.
* The first line of the file must contain a comma-separated list of column names.
* The url-value can be a resource or a file on the system.

Each line of this file must contain data that is interpreted as values for each column name. When each line is read from the file, it's possible to perform the necessary processing to create or merge data into the graph.

As CSV file usually represent either node or relationship lists and to create nodes and relationships separately it's necessary to parse several times that file.

Example of csv file:

Explain

id,title,country,year,summary

1,Wall Street,USA,1987, Every dream has a price.

2,The American President,USA,1995, Why can't the most powerful man in the world have the one thing he wants most?

3,The Shawshank Redemption,USA,1994, Fear can hold you prisoner. Hope can set you free.

Before loading data from CSV files into graph, we need to confirm that the data retrieved looks ok. To do this we can first print the lines of the file and get some information about the data to be loaded.

Example:

LOAD CSV WITH HEADERS

FROM 'http://data.neo4j.com/intro-neo4j/movies\_to\_load.csv'

as LINE

RETURN count(\*)

Or even better we probably want to see how the data’s are stored:

LOAD CSV WITH HEADERS

FROM 'http://data.neo4j.com/intro-neo4j/movies\_to\_load.csv'

AS line

RETURN \* LIMIT 1

Since all the lines are with the same format we can limit the visualization to 1.

The output of the last query should be:

Explain

line

{

"summary": " Every dream has a

price.",

"country": "USA",

"year": "1987",

"id": "1",

"title": "Wall Street"

}

Notice that the *summary* column's data has an extra space before the data in the file, to make sure not to load data like this we need to beautify the output:

LOAD CSV WITH HEADERS

FROM 'http://data.neo4j.com/intro-neo4j/movies\_to\_load.csv'

AS line

RETURN line.id, line.title, toInteger(line.year), trim(line.summary)

The output will be formatted to this:

| **line.id** | **line.title** | **toInteger(line.year)** | **trim(line.summary)** |
| --- | --- | --- | --- |
| "1" | "Wall Street" | 1987 | "Every dream has a price." |
| "2" | "The American President" | 1995 | "Why can't the most powerful man in the world have the one thing he wants most?" |
| "3" | "The Shawshank Redemption" | 1994 | "Fear can hold you prisoner. Hope can set you free." |

Now we can create nodes and relationships from this file:

LOAD CSV WITH HEADERS

FROM 'http://data.neo4j.com/intro-neo4j/movies\_to\_load.csv'

AS line

CREATE (movie:Movie {movieId: line.id, title: line.title, released: toInteger(line.year), tagline: trim(line.summary)})

Another example can be the file that holds the data of Person:

Explain

Id,name,birthyear

1,Charlie Sheen, 1965

2,Oliver Stone, 1946

3,Michael Douglas, 1944

4,Martin Sheen, 1940

5,Morgan Freeman, 1937

In this case we need to pay more attention at the people in the database that have already this name to prevent duplications. Instead of creating them, we need to use the **MERGE** to ensure unique entries after the import.

LOAD CSV WITH HEADERS

FROM 'http://data.neo4j.com/intro-neo4j/person\_to\_load.csv'

AS line

MERGE (actor:Person {personId: line.Id})

ON CREATE

SET actor.name = line.name, actor.born = toInteger(trim(line.birthyear))

Now let's create relationships between the movies and the actors using the csv file:

Explain

personId,movieId,role

1,1,Bud Fox

4,1,Carl Fox

3,1,Gordon Gekko

4,2,A.J. MacInerney

3,2,President Andrew Shepherd

5,3,Ellis Boyd 'Red' Redding

The relative query will be:

LOAD CSV WITH HEADERS

FROM 'http://data.neo4j.com/intro-neo4j/roles\_to\_load.csv'

AS line

MATCH (movie:Movie { movieId: line.movieId })

MATCH (person:Person { personId: line.personId })

CREATE (person)-[:ACTED\_IN { roles: [line.role]}]->(movie)