

# neo4j

January 6, 2021

```
[1]: from IPython.display import Image
```

## 0.0.1 Introduction to Graph Databases

### What is a graph database?

- Like other databases, a graph database is an database management system with Create, Read, Update and Delete (CRUD).
- Generally built for use with online transaction processing (OLTP) systems.
- Unlike other databases, relationships take first priority in graph databases. This means your application doesn't have to infer data connections using foreign keys or out-of-band processing, such as MapReduce.

**What is a graph?** A graph is composed of two elements: nodes and relationships.

Each node represents an entity (a person, place, thing, category or other piece of data). With Neo4j, nodes can have **labels** that are used to define types for nodes. For example, a *Location* node is a node with the label *Location*. That same node can also have a label, *Residence*. Another *Location* node can also have a label, *Business*. A label can be used to group nodes of the same type. For example, you may want to retrieve all of the *Business* nodes.

```
[2]: Image('images/nodes.png', width=400)
```

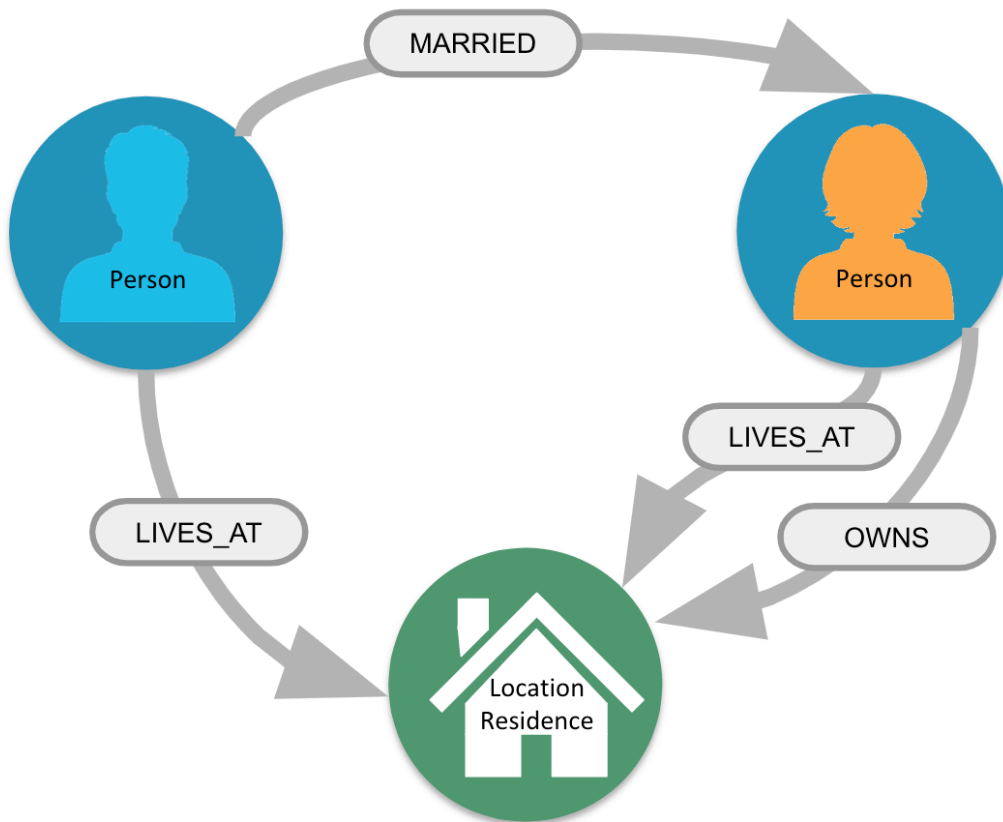
[2]:



Each relationship represents how two nodes are connected. For example, the two nodes *Person* and *Location*, might have the relationship *LIVES\_AT* pointing from a *Person* node to *Location* node. A relationship represents the verb or action between two entities. The *MARRIED* relationship is defined from one *Person* node to another *Person* node. Although the relationship is defined as directional, it can be queried in a non-directional manner. That is, you can query if two *Person* nodes have a *MARRIED* relationship, regardless of the direction of the relationship. For some data models, the direction of the relationship is significant. For example, in Facebook, using the *KNOWS* relationship is used to indicate which *Person* invited the other *Person* to be a friend.

```
[3]: Image('images/relationships.png', width=400)
```

```
[3]:
```

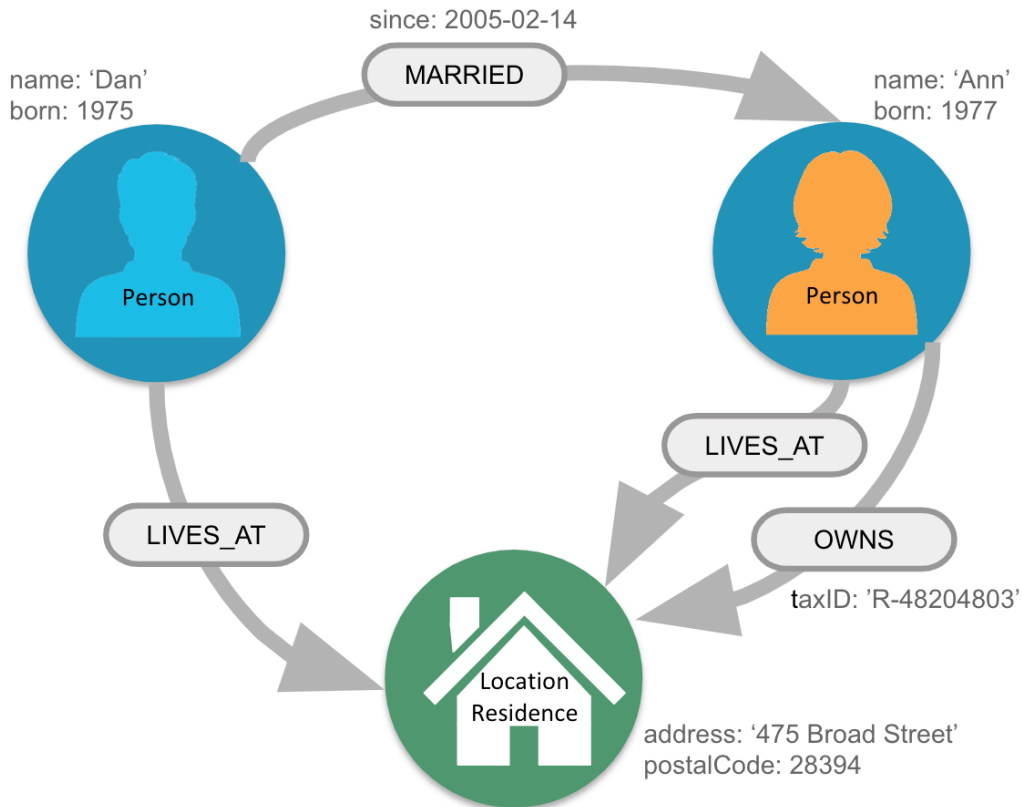


This general-purpose structure allows you to model all kinds of scenarios: from a system of roads, to a network of devices, to a population's medical history, or anything else defined by relationships.

The Neo4j database is a property graph. You can add properties to nodes and relationships to further enrich the graph model.

```
[4]: Image('images/properties.png', width=400)
```

```
[4]:
```



```
[5]: Image('images/comparison_with_relational_database.png')
```

[5]: There are some similarities between a relational model and a graph model,

Relational	Graph
Rows	Nodes
Joins	Relationships
Table names	Labels
Columns	Properties

But, there are some ways in which the relational model differs from the graph model,

Relational	Graph
Each column must have a 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.

How you model data from relational vs graph differs:

Relational	Graph
Try and 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 i.e. focus on classes rather than objects.	Common to use actual data items when modeling.

**QUIZ Q1)** What elements make up a graph?

- ☐ tuples
- nodes
- ☐ documents
- relationships

**Q2)** Suppose that you want to create a graph to model customers, products, what products a customer buys, and what products a customer rated. You have created nodes in the graph to represent the customers and products. In this graph, what relationships would you define?

- BOUGHT
- ☐ IS\_A\_CUSTOMER
- ☐ IS\_A\_PRODUCT
- RATED

**Q3)** What query language is used with a Neo4j Database?

- ☐ SQL
- ☐ CQL
- Cypher
- ☐ OPath

### 0.0.2 Introduction to Neo4j

**Neo4j Database** The heart of the Neo4j Graph Platform is the Neo4j Database. The Neo4j Graph Platform includes out-of-the-box tooling that enables you to access graphs in Neo4j Databases. In addition, Neo4j provides APIs and drivers that enable you to create applications and custom tooling for accessing and visualizing graphs.

**Neo4j Database: Index-free adjacency** 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 (i.e. it has a graph as its core data model), it supports very large graphs where connected data can be traversed in constant time without the need for an index.

**Neo4j Database: ACID (Atomic, Consistent, Isolated, Durable)** Transactionality is very important for robust applications that require an ACID (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** 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 retrieve data, whether it is on disk, or cached in memory. The graph engine has been improved with every release of Neo4j to provide the most efficient access to an application's graph data. There are many ways that you can tune the performance of the engine to suit your particular application needs.

**Language and driver support** Because Neo4j is open source, you can delve into the details of how the Neo4j Database is accessed, but most developers simply use Neo4j without needing a deeper understanding of the underlying code. Neo4j provides a full stack that implements all levels of access to the database and clustering layer where you can use our published APIs. The language used for querying the Neo4j database is Cypher, an open source language.

In addition, Neo4j supports Java, JavaScript, Python, C#, and Go drivers out of the box 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 as encrypting the data. For example, you can write 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.

It is also possible for you to develop your own server-side extensions in Java that access the data in the database directly without using Cypher. The Neo4j community has developed drivers for a number of languages including Ruby, PHP, and R. You can also extend the functionality of Neo4j by creating user defined functions and procedures that are callable from Cypher.

**Libraries** Neo4j has a published, open source Cypher library, **Awesome Procedures on Cypher (APOC)** that contain many useful procedures you can call from Cypher. Another Cypher library is the **Graph Algorithms** library, shown here, that can help you to analyze data in your graphs. Graph analytics are important because with Neo4j, the technology can expose questions about the data that you never thought to ask. And finally, you can use the **GraphQL** library (tree-based subset of a graph) to access a Neo4j Database. These libraries are available as plug-ins to your Neo4j development environment, but there are many other libraries that have been written by users for accessing Neo4j.

**Tools** In a development environment, you will use the **Neo4j Browser** or a **Web browser** to access data and test your Cypher statements, most of which will be used as part of your application code. Neo4j Browser is an application that uses the JavaScript Bolt driver to access the graph engine of the Neo4j database server. Neo4j also has a new tool called **Bloom** that enables you to visualize a graph without knowing much about Cypher. In addition, there are many tools for importing and exporting data between flat files and a Neo4j Database, as well as an ETL tool.

**QUIZ Q1)** What are some of the benefits provided by the Neo4j Graph Platform?

- Database clustering
- ACID
- Index free adjacency
- Optimized graph engine

**Q2)** What libraries are available for the Neo4j Graph Platform?

- APOC
- ☐ JGraph
- Graph Algorithms
- GraphQL

**Q3)** What are some of the language drivers that come with Neo4j out of the box?

- Java

- Ruby
- Python
- Javascript

```
[6]: from IPython.display import Image
```

### 0.0.3 Introduction to Cypher

**Cypher is ASCII art** Being a declarative language, Cypher focuses on the clarity of expressing what to retrieve from a graph, not on how to retrieve it. You can think of Cypher as mapping English language sentence structure to patterns in a graph. For example, the nouns are nodes of the graph, the verbs are the relationships in the graph, and the adjectives and adverbs are the properties.

**Nodes** Cypher uses a pair of parentheses like `()`, `(n)` to represent a node, much like a circle on a whiteboard. Recall that a node typically represents an entity in your domain. An anonymous node, `()`, represents one or more nodes during a query processing where there are no restrictions of the type of node or the properties of the node. When you specify `(n)` for a node, you are telling the query processor that for this query, use the variable `n` to represent nodes that will be processed later in the query for further query processing or for returning values from the query.

**Labels** Nodes in a graph are typically labeled. Labels are used to group nodes and filter queries against the graph. That is, labels can be used to optimize queries. In the Movie database you will be working with, the nodes in this graph are labeled `Movie` or `Person` to represent two types of nodes.

You can filter the types of nodes that you are querying, by specifying a label for a node. A node can have zero or more labels.

Here are simplified syntax examples for specifying a node,

```
()
(variable)
(:Label)
(variable:Label)
(:Label1:Label2)
(variable:Label1:Label2)
```

Notice that a node must have the parentheses. The labels and the variable for a node are optional.

Here are examples of specifying nodes in Cypher,

```
()           // anonymous node not be referenced later in the query
(p)          // variable p, a reference to a node used later
(:Person)    // anonymous node of type Person
(p:Person)   // p, a reference to a node of type Person
(p:Actor:Director) // p, a reference to a node of types Actor and Director
```

A node can have multiple labels. For example a node can be created with a label of `Person` and that same node can be modified to also have the label of `Actor` and/or `Director`.

In Cypher, you can place a comment (starts with `//`) anywhere in your Cypher to specify that the rest of the line is interpreted as a comment.

**Examining the data model** When you are first learning about the data (nodes, labels, etc.) in a graph, it is helpful to examine the data model of the graph. You do so by executing `CALL db.schema`, which calls the Neo4j procedure that returns information about the nodes, labels, and relationships in the graph.

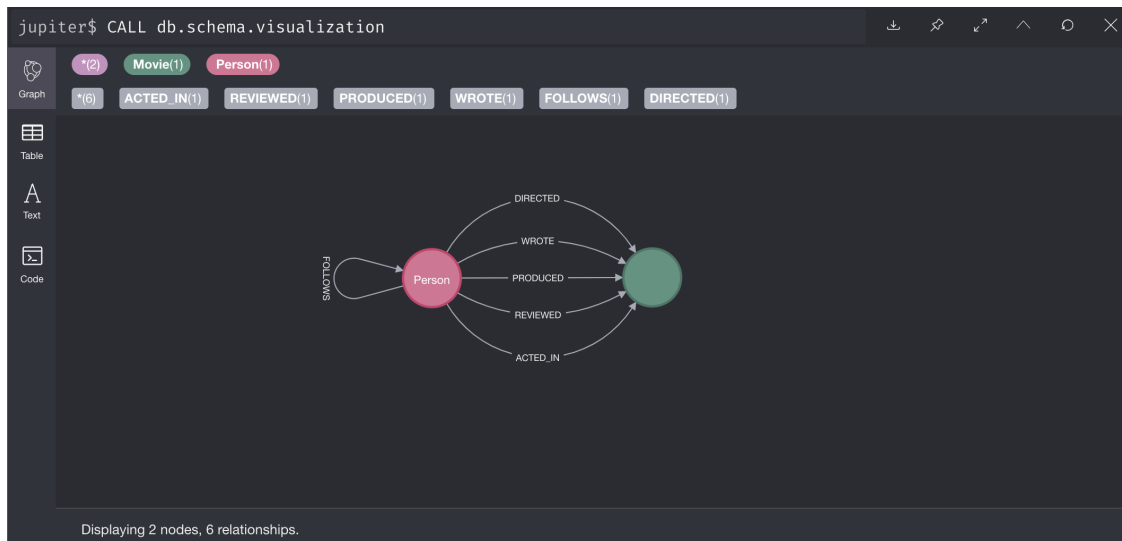
Please notice that, you should use `CALL db.schema.visualization` instead of `CALL db.schema` in the new versions of neo4j.

If you want to see the result, you can follow these steps, 1. `:play Movie Graph` 2. Go to page 2 in the pop-up after running the cypher in the step 1. 3. Run the create statement as you can see in the page. 4. Finally, you can run the cypher `CALL db.schema.visualization`

After these steps, you can see the result as following,

```
[7]: Image('images/db_schema.png', width=800)
```

[7]:



**Using MATCH to retrieve nodes** The most widely used Cypher clause is `MATCH`. The `MATCH` clause 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. As part of query, you can return nodes or data from the nodes using the `RETURN` clause. The `RETURN` clause must be the last clause of a query to the graph.

Here are simplified syntax examples for a query,

```
MATCH (variable)
RETURN variable
```

```
MATCH (variable:Label)
RETURN variable
```



Notice that the Cypher keywords **MATCH** and **RETURN** are upper-case. This coding convention is described in the Cypher Style Guide and will be used in this training. This **MATCH** clause returns all nodes in the graph, where the optional Label is used to return a subgraph if the graph contains nodes of different types. The variable must be specified here, otherwise the query will have nothing to return.

Here are example queries to the Movie database,

```
// returns all nodes in the graph
MATCH (n)
RETURN n

// returns all person nodes in the graph
MATCH (p:Person)
RETURN p
```

**!!!** 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 retrieve the nodes which will perform better than not using a label for the **MATCH**.

**Properties** In Neo4j, a node (and a relationship, which you will learn about later) can have properties that are used to further define a node. A property is identified by its property key. Recall that nodes are used to represent the entities of your business model. A property is defined for a node and not for a type of node. All nodes of the same type need not have the same properties.

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.

**Examining property keys** As you prepare to create Cypher queries that use property values to filter a query, you can view the values for property keys of a graph by simply clicking the Database icon in Neo4j Browser. Alternatively, you can execute **CALL db.propertyKeys**, which calls the Neo4j library method that returns the property keys for the graph.

**Retrieving nodes filtered by a property value** You have learned previously that you can filter node retrieval by specifying a label. Another way you can filter a retrieval is to specify a value for a property. Any node that matches the value will be retrieved.

Here are simplified syntax examples for a query where we specify one or more values for properties that will be used to filter the query results and return a subset of the graph,

```
MATCH (variable {propertyKey: propertyValue})
RETURN variable

MATCH (variable:Label {propertyKey: propertyValue})
RETURN variable

MATCH (variable {propertyKey1: propertyValue1, propertyKey2: propertyValue2})
RETURN variable

MATCH (variable:Label {propertyKey: propertyValue, propertyKey2: propertyValue2})
RETURN variable
```

Here is an example where we filter the query results using a property value. We only retrieve Person nodes that have a born property value of 1970,

```
MATCH (p:Person {born: 1970})
RETURN p
```

Here is an example where we specify two property values for the query,

```
MATCH (m:Movie {released: 2003, tagline: 'Free your mind'})
RETURN m
```

And the result of the query as follows in table format,

[8]: `Image('images/property_result.png', width=800)`



**Returning property values** Here are simplified syntax examples for returning property values, rather than nodes,

```
MATCH (variable {prop1: value})
RETURN variable.prop2
```

```
MATCH (variable:Label {prop1: value})
RETURN variable.prop2
```

```
MATCH (variable:Label {prop1: value, prop2: value})
RETURN variable.prop3
```

```
MATCH (variable {prop1:value})
RETURN variable.prop2, variable.prop3
```

In this example, we use the born property to refine the query, but rather than returning the nodes, we return the name and born values for every node that satisfies the query.

```
MATCH (p:Person {born: 1965})
RETURN p.name, p.born
```

**Specifying aliases** If you want to customize the headings for a table containing property values, you can specify aliases for column headers.

Here is the simplified syntax for specifying an alias for a property value,

```
MATCH (variable:Label {propertyKey1: propertyValue1})
RETURN variable.propertyKey2 AS alias2
```

Here we specify aliases for the returned property values,

```
MATCH (p:Person {born: 1965})
RETURN p.name AS name, p.born AS `birth year`
```

!!! If you want a heading to contain a space between strings, you must specify the alias with the back tick character, rather than a single or double quote character. In fact, you can specify any variable, label, relationship type, or property key with a space also by using the back tick character.

**Relationships** Relationships are what make Neo4j graphs such a powerful tool for connecting complex and deep data. 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 graph where you want to retrieve nodes, you can use relationships between nodes to filter a query.

Here is how Cypher uses ASCII art to specify path used for a query,

```
()          // 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
```

**Querying using relationships** In your MATCH clause, you specify how you want a relationship to be used to perform the query. The relationship can be specified with or without direction.

Here are simplified syntax examples for retrieving a set of nodes that satisfy one or more directed and typed relationships,

```
MATCH (node1)-[:REL_TYPE]->(node2)
RETURN node1, node2

MATCH (node1)-[:REL_TYPEA | :REL_TYPEB]->(node2)
RETURN node1, node2
```

**Using a relationship in a query** Here is an example where we retrieve the Person nodes that have the ACTED\_IN relationship to the Movie, The Matrix. In other words, show me the actors that acted in The Matrix.

```
MATCH (p:Person)-[rel:ACTED_IN]->(m:Movie {title: 'The Matrix'})
RETURN p, rel, m
```

For this query, we are using the variable *p* to represent the *Person* nodes during the query, the variable *m* to represent the *Movie* node retrieved, and the variable *rel* to represent the *relationship* for the relationship type, *ACTED\_IN*. We return a graph with the *Person* nodes, the *Movie* node and their *ACTED\_IN* relationships.

**Important:** You specify node labels whenever possible in your queries as it optimizes the retrieval in the graph engine. That is, you should not specify this same query as,

```
MATCH (p)-[rel:ACTED_IN]->(m {title:'The Matrix'})
RETURN p,m
```

**Querying by multiple relationships** Here is another example where we want to know the movies that Tom Hanks acted in and directed,

```
MATCH (p:Person {name: 'Tom Hanks'})-[:ACTED_IN|:DIRECTED]->(m:Movie)
RETURN p.name, m.title
```

Suppose you wanted to retrieve the actors that acted in *The Matrix*, but you do not need any information returned about the Movie node. You need not specify a variable for a node in a query if that node is not returned or used for later processing in the query. You can simply use the anonymous node in the query as follows,

```
MATCH (p:Person)-[:ACTED_IN]->(:Movie {title: 'The Matrix'})
RETURN p.name
```

**Using an anonymous relationship for a query** Suppose you want to find all people who are in any way connected to the movie, *The Matrix*. You can specify an empty relationship type in the query so that all relationships are traversed and the appropriate results are returned. In this example, we want to retrieve all *Person* nodes that have any type of connection to the *Movie* node, with the title, *The Matrix*. This query returns more nodes with the relationships types, *DIRECTED*, *ACTED\_IN*, and *PRODUCED*.

```
MATCH (p:Person)-->(m:Movie {title: 'The Matrix'})
RETURN p, m
```

Here are other examples of using the anonymous relationship,

```
MATCH (p:Person)--(m:Movie {title: 'The Matrix'})
RETURN p, m
```

```
MATCH (m:Movie)<--(p:Person {name: 'Keanu Reeves'})
RETURN p, m
```

**Retrieving the relationship types** There is a built-in function, `type()` that returns the relationship type of a relationship.

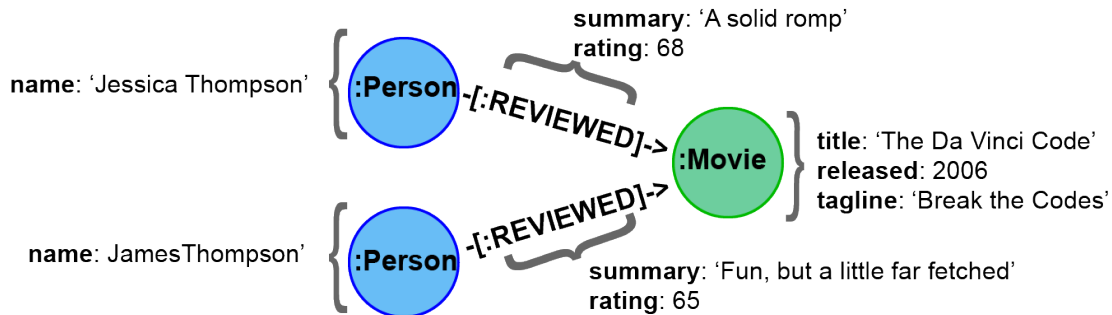
```
MATCH (p:Person)-[rel]->(:Movie {title:'The Matrix'})
RETURN p.name, type(rel)
```

**Retrieving properties for relationships** Recall that a node can have a set of properties, each identified by its property key. Relationships can also have properties. This enables your graph model to provide more data about the relationships between the nodes.

Here is an example from the *Movie graph*. The movie, *The Da Vinci Code* has two people that reviewed it, *Jessica Thompson* and *James Thompson*. Each of these *Person* nodes has the *REVIEWED* relationship to the *Movie* node for *The Da Vinci Code*. Each relationship has properties that further describe the relationship using the *summary* and *rating* properties.

```
[9]: Image('images/reviewed_properties.png', width=800)
```

[9]:



Just as you can specify property values for filtering nodes for a query, you can specify property values for a relationship. This query returns the name of the person who gave the movie a rating of 65,

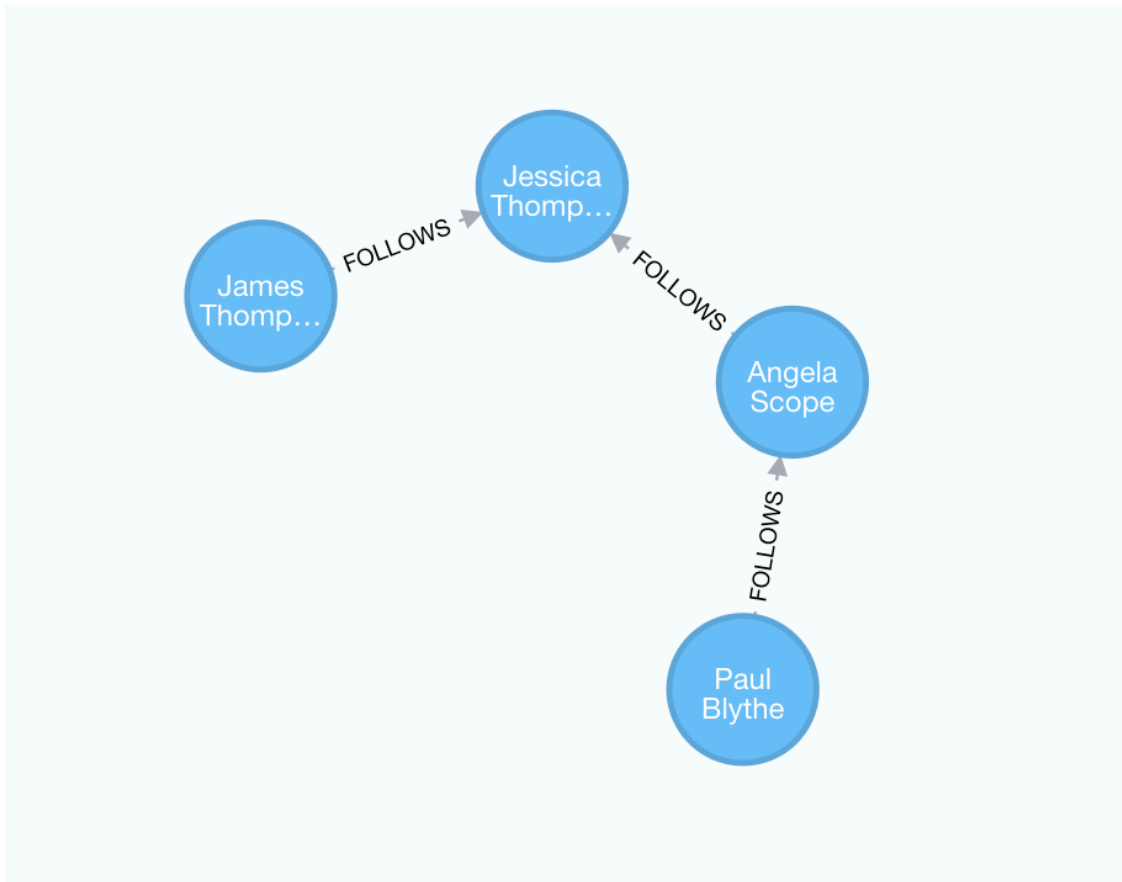
```
MATCH (p:Person)-[:REVIEWED {rating: 65}]->(:Movie {title: 'The Da Vinci Code'})
RETURN p.name
```

**Using patterns for queries** Since relationships are directional, it is important to understand how patterns are used in graph traversal during query execution. How a graph is traversed for a query depends on what directions are defined for relationships and how the pattern is specified in the `MATCH` clause.

Here is an example of where the *FOLLOWS* relationship is used in the Movie graph. Notice that this relationship is directional.

```
[10]: Image('images/follows_relationships.png', width=400)
```

[10]:

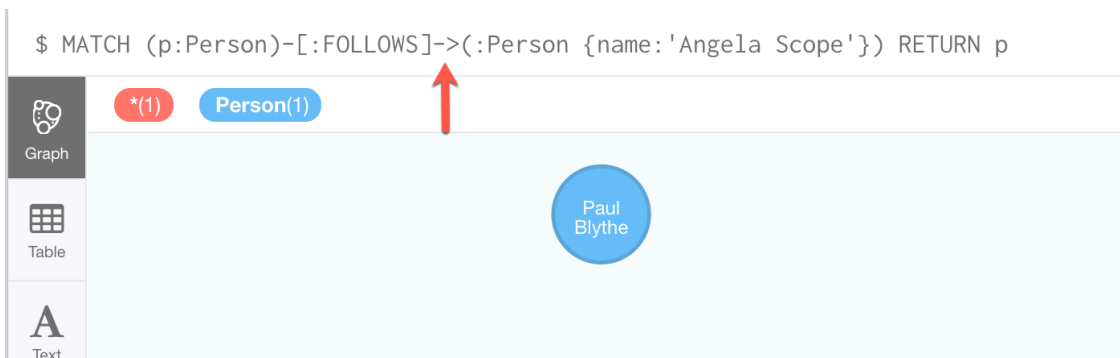


We can perform a query that returns all *Person* nodes who follow *Angela Scope*:

```
MATCH (p:Person)-[:FOLLOWS]->(:Person {name:'Angela Scope'})
RETURN p
```

[11]: `Image('images/angela_followers.png', width=400)`

[11]:

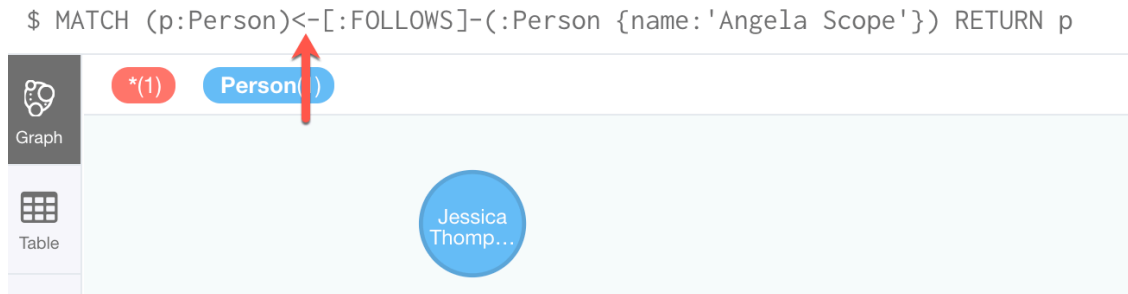


If we reverse the direction in the pattern, the query returns different results,

```
MATCH (p:Person)<-[:FOLLOWS]-(:Person {name:'Angela Scope'})
RETURN p
```

```
[12]: Image('images/followed_by_angela.png', width=400)
```

[12]:



**Traversing relationships** Since we have a graph, we can traverse through nodes to obtain relationships further into the traversal.

For example, we can write a Cypher query to return all followers of the followers of *Jessica Thompson*.

```
MATCH (p:Person)-[:FOLLOWS]->(:Person)-[:FOLLOWS]->(:Person {name: 'Jessica Thompson'})
RETURN p
```

This query could also be modified to return each person along the path by specifying variables for the nodes and returning them. In addition, you can assign a variable to the path and return the path as follows,

```
MATCH path = (:Person)-[:FOLLOWS]->(:Person)-[:FOLLOWS]->(:Person {name: 'Jessica Thompson'})
RETURN path
```

**Using relationship direction to optimize a query (IMPORTANT)** When querying the relationships in a graph, you can take advantage of the direction of the relationship to traverse the graph. For example, suppose we wanted to get a result stream containing rows of actors and the movies they acted in, along with the director of the particular movie.

Here is the Cypher query to do this. Notice that the direction of the traversal is used to focus on a particular movie during the query,

```
MATCH (a:Person)-[:ACTED_IN]->(m:Movie)<-[:DIRECTED]-(d:Person)
RETURN a.name, m.title, d.name
```

Here are the Neo4j-recommended Cypher coding standards that we use in this training,

- Node labels are CamelCase and begin with an upper-case letter (examples: *Person*, *NetworkAddress*). Note that node labels are case-sensitive.

- Property keys, variables, parameters, aliases, and functions are camelCase and begin with a lower-case letter (examples: *businessAddress*, *title*). Note that these elements are case-sensitive.
- Relationship types are in upper-case and can use the underscore. (examples: *ACTED\_IN*, *FOLLOWS*). Note that relationship types are case-sensitive and that you cannot use the “-” character in a relationship type.
- Cypher keywords are upper-case (examples: *MATCH*, *RETURN*). Note that Cypher keywords are case-insensitive, but a best practice is to use upper-case.
- String constants are in single quotes, unless the string contains a quote or apostrophe (examples: *‘The Matrix’*, *“Something’s Gotta Give”*). Note that you can also escape single or double quotes within strings that are quoted with the same using a backslash character.
- Specify variables only when needed for use later in the Cypher statement.
- Place named nodes and relationships (that use variables) before anonymous nodes and relationships in your *MATCH* clauses when possible.
- Specify anonymous relationships with *-->*, *--*, or *<--*

**QUIZ Q1)** Suppose you have a graph that contains nodes representing customers and other business entities for your application. The node label in the database for a customer is *Customer*. Each *Customer* node has a property named *email* that contains the customer’s email address. What Cypher query do you execute to return the email addresses for all customers in the graph?

- ☐ *MATCH (n) RETURN n.Customer.email*
- ☒ *MATCH (c:Customer) RETURN c.email*
- ☐ *MATCH (Customer) RETURN email*
- ☐ *MATCH (c) RETURN Customer.email*

**Q2)** Suppose you have a graph that contains *Customer* and *Product* nodes. A *Customer* node can have a *BOUGHT* relationship with a *Product* node. *Customer* nodes can have other relationships with *Product* nodes. A *Customer* node has a property named *customerName*. A *Product* node has a property named *productName*. What Cypher query do you execute to return all of the products (by name) bought by customer ‘ABCCO’.

- ☐ *MATCH (c:Customer {customerName: 'ABCCO'}) RETURN c.BOUGHT.productName*
- ☐ *MATCH (:Customer 'ABCCO')-[:BOUGHT]->(p:Product) RETURN p.productName*
- ☐ *MATCH (p:Product)<-[:BOUGHT\_BY]-(:Customer 'ABCCO') RETURN p.productName*
- ☒ *MATCH (:Customer {customerName: 'ABCCO'})-[:BOUGHT]->(p:Product) RETURN p.productName*

**Q3)** When must you use a variable in a *MATCH* clause?

- ☐ When you want to query the graph using a node label.
- ☐ When you specify a property value to match the query.
  - When you want to use the node or relationship to return a result.
- ☐ When the query involves 2 types of nodes.

```
[13]: from IPython.display import display, Image
```



#### 0.0.4 Getting More Out of Queries

**Filtering queries using WHERE** The format for filtering you have learned thus far only tests equality, where you must specify values for the properties to test with. What if you wanted more flexibility about how the query is filtered? For example, you want to retrieve all movies released after 2000, or retrieve all actors born after 1970 who acted in movies released before 1995. Most applications need more flexibility in how data is filtered.

The most common clause use to filter queries is the **WHERE** clause that follows a **MATCH** clause. In the **WHERE** clause, you can place conditions that are evaluated at runtime to filter the query.

Previous way,

```
MATCH (p:Person)-[:ACTED_IN]->(m:Movie {released: 2008})
RETURN p, m
```

Here is one way you specify the same query using the **WHERE** clause,

```
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WHERE m.released = 2008
RETURN p, m
```

Here are the examples using the **WHERE** clause,

```
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WHERE m.released = 2008 OR m.released = 2009
RETURN p, m
```

Specifying ranges in **WHERE** clauses,

```
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WHERE m.released >= 2003 AND m.released <= 2004
RETURN p.name, m.title, m.released
```

Here is the same query different way,

```
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WHERE 2003 <= m.released <= 2004
RETURN p.name, m.title, m.released
```

Here are the examples about the **WHERE** clause in the **Movie** graph,

```
// returns all person nodes in the graph
MATCH (p)
WHERE p:Person
RETURN p.name

// returns all person nodes acted in the movie named as The Matrix
MATCH (p)-[:ACTED_IN]->(m)
WHERE p:Person AND m:Movie AND m.title = 'The Matrix'
RETURN p.name
```

**Testing the existence of a property** Suppose we only want to return the movies that the actor, Jack Nicholson acted in with the condition that they must all have a tagline,

```
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WHERE p.name = 'Jack Nicholson' AND exists(m.tagline)
RETURN m.title, m.tagline
```

**Testing strings** Cypher has a set of string-related keywords that you can use in your WHERE clauses to test string property values. You can specify **STARTS WITH**, **ENDS WITH**, and **CONTAINS**.

```
MATCH (p:Person)-[:ACTED_IN]->()
WHERE p.name STARTS WITH 'Michael'
RETURN p.name
```

Note that the comparison of strings is **case-sensitive**. There are a number of string-related functions you can use to further test strings. For example, if you want to test a value, regardless of its case, you could call the `toLower()` function to convert the string to lower case before it is compared.

```
MATCH (p:Person)-[:ACTED_IN]->()
WHERE toLower(p.name) STARTS WITH 'michael'
RETURN p.name
```

**Important:** In this example where we are converting a property to lower case, if an index has been created for this property, it will not be used at runtime.

**Testing with regular expressions** If you prefer, you can test property values using regular expressions. You use the syntax `=~` to specify the regular expression you are testing with. Here is an example where we test the name of the **Person** using a regular expression to retrieve all **Person** nodes with a name property that begins with Tom,

```
MATCH (p:Person)
WHERE p.name = ~'Tom.*'
RETURN p.name
```

**Testing with patterns** Suppose we want to find persons who wrote the movie but didn't direct it,

```
MATCH (p:Person)-[:WROTE]->(m:Movie)
WHERE NOT exists((p)-[:DIRECTED]->(m))
RETURN p.name, m.title
```

Here is another example where we want to find Gene Hackman and the movies that he acted in with another person who also directed the movie,

```
MATCH (gene:Person)-[:ACTED_IN]->(m:Movie)<-[:ACTED_IN]-(other:Person)
WHERE gene.name = 'Gene Hackman'
AND exists((other)-[:DIRECTED]->(m))
RETURN gene, other, m
```

**Testing with list values** You can define the list in the WHERE clause. During the query, the graph engine will compare each property with the values **IN** the list. You can place either numeric or string values in the list, but typically, elements of the list are of the same type of data. If you are testing with a property of a string type, then all the elements of the list should be strings.

```

MATCH (p:Person)
WHERE p.born IN [1965, 1970]
RETURN p.name as name, p.born as yearBorn

```

You can also compare a value to an existing list in the graph.

We know that the `:ACTED_IN` relationship has a property, `roles` that contains the list of roles an actor had in a particular movie they acted in. Here is the query we write to return the name of the actor who played *Neo* in the movie *The Matrix*,

```

MATCH (p:Person)-[r:ACTED_IN]->(m:Movie)
WHERE 'Neo' IN r.roles AND m.title = 'The Matrix'
RETURN p.name

```

**Specifying multiple MATCH patterns** This MATCH clause includes a pattern specified by two paths separated by a comma,

```

MATCH (a:Person)-[:ACTED_IN]->(m:Movie),
      (m:Movie)<-[:DIRECTED]-(d:Person)
WHERE m.released = 2000
RETURN a.name, m.title, d.name

```

If possible, you should write the same query as follows,

```

MATCH (a:Person)-[:ACTED_IN]->(m:Movie)<-[:DIRECTED]-(d:Person)
WHERE m.released = 2000
RETURN a.name, m.title, d.name

```

**Using two MATCH patterns** Here are some examples of specifying two paths in a MATCH clause. In the first example, we want the actors that worked with *Keanu Reeves* to meet *Hugo Weaving*, who has worked with *Keanu Reeves*. Here we retrieve the actors who acted in the same movies as *Keanu Reeves*, but not when *Hugo Weaving* acted in the same movie. To do this, we specify two paths for the MATCH,

```

MATCH (keanu:Person)-[:ACTED_IN]->(movie:Movie)<-[:ACTED_IN]-(n:Person),
      (hugo:Person)
WHERE keanu.name = 'Keanu Reeves' AND
      hugo.name = 'Hugo Weaving'
AND NOT (hugo)-[:ACTED_IN]->(movie)
RETURN n.name

```

Here is another example where two patterns are necessary. Suppose we want to retrieve the movies that *Meg Ryan* acted in and their respective directors, as well as the other actors that acted in these movies. Here is the query to do this,

```

MATCH (meg:Person)-[:ACTED_IN]->(m:Movie)<-[:DIRECTED]-(d:Person),
      (other:Person)-[:ACTED_IN]->(m)
WHERE meg.name = 'Meg Ryan'
RETURN m.title as movie, d.name AS director , other.name AS `co-actors`

```

You can find the result of last cypher as follows,

```
[14]: Image('images/meg_coactors.png', width=600)
```

```
[14]:
```



	movie	director	co-actors
1	"Sleepless in Seattle"	"Nora Ephron"	"Victor Garber"
2	"Sleepless in Seattle"	"Nora Ephron"	"Tom Hanks"
3	"Sleepless in Seattle"	"Nora Ephron"	"Bill Pullman"
4	"Sleepless in Seattle"	"Nora Ephron"	"Rita Wilson"
5	"Sleepless in Seattle"	"Nora Ephron"	"Rosie O'Donnell"
6	"You've Got Mail"	"Nora Ephron"	"Tom Hanks"
7	"You've Got Mail"	"Nora Ephron"	"Tom Hanks"

Started streaming 20 records after 2 ms and completed after 11 ms.

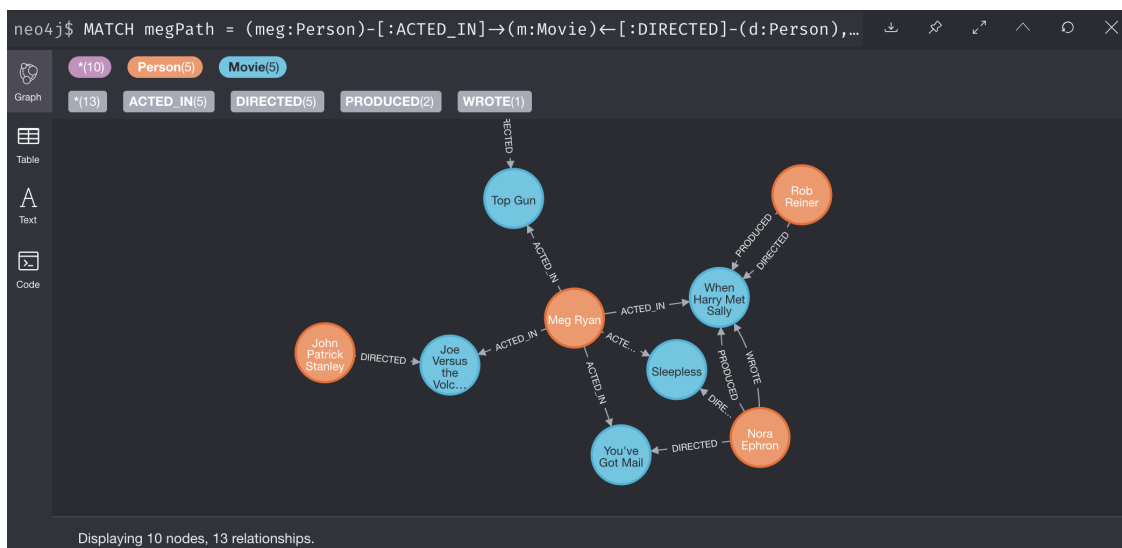
**Setting path variables** What if we want to see the result of above query as graph?

```
MATCH megPath = (meg:Person)-[:ACTED_IN]->(m:Movie)<-[:DIRECTED]-(d:Person),  
              (other:Person)-[:ACTED_IN]->(m)  
WHERE meg.name = 'Meg Ryan'  
RETURN megPath
```

Here is the result,

```
[15]: Image('images/meg_path.png', width=600)
```

```
[15]:
```



**Specifying varying length paths** Any graph that represents social networking, trees, or hierarchies will most likely have multiple paths of varying lengths. Think of the *connected* relationship in *LinkedIn* and how connections are made by people connected to more people. The *Movie* database for this training does not have much depth of relationships, but it does have the *:FOLLOWS* relationship that you learned about earlier. You write a **MATCH** clause where you want to find all of the followers of the followers of a *Person* by specifying a numeric value for the number of hops in the path. Here is an example where we want to retrieve all *Person* nodes that are exactly two hops away,

```
MATCH (follower:Person)-[:FOLLOWS*2]->(p:Person)
WHERE follower.name = 'Paul Blythe'
RETURN p
```

```
[16]: Image('images/two_hop_relationship.png', width=600)
```



If we had specified *:FOLLOWS* rather than *:FOLLOWS2*, the query would return all *Person* nodes that are in the *:FOLLOWS\** path from *Paul Blythe*.

Here are simplified syntax examples for how varying length patterns are specified in Cypher.

- 1) Retrieve all paths of any length with the relationship, *:RELTYPE* from *nodeA* to *nodeB* and beyond,

```
(nodeA)-[:RELTYPE*]->(nodeB)
```

- 2) Retrieve all paths of any length with the relationship, *:RELTYPE* from *nodeA* to *nodeB* or from *nodeB* to *nodeA* and beyond. This is usually a very expensive query so you should place limits on how many nodes are retrieved,

```
(nodeA)-[:RELTYPE*]-(nodeB)
```

- 3) Retrieve the paths of length 3 with the relationship, *:RELTYPE* from *nodeA* to *nodeB*,

```
(node1)-[:RELTYPE*3]->(node2)
```

- 4) Retrieve the paths of lengths 1, 2, or 3 with the relationship, *:RELTYPE* from *nodeA* to *nodeB*, *nodeB* to *nodeC*, as well as, *nodeC* to *\*\_nodeD\** (up to three hops),

```
(node1)-[:RELTYPE*1..3]->(node2)
```

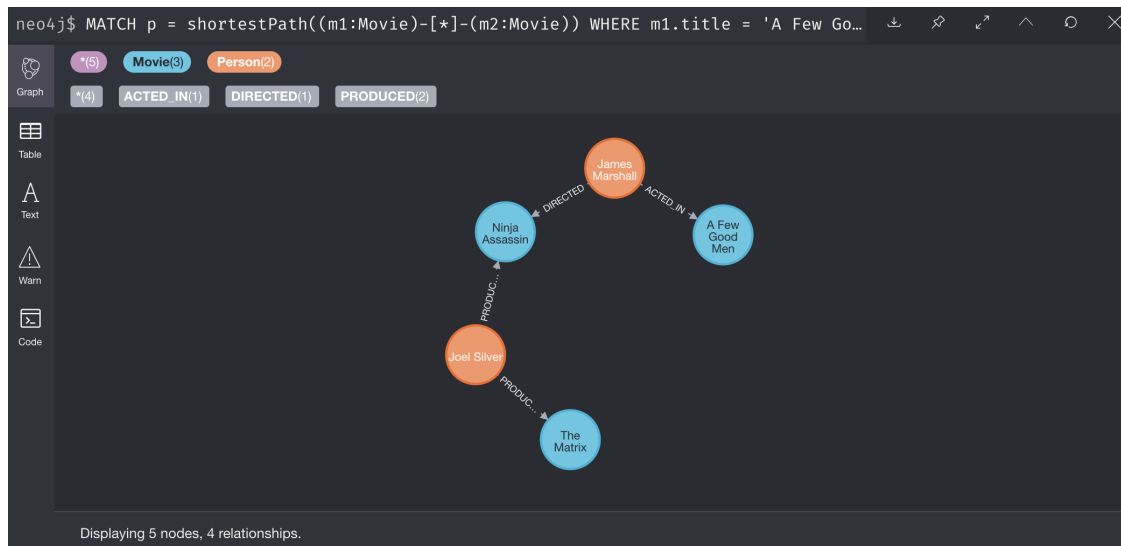
**Finding the shortest path** A built-in function that you may find useful in a graph that has many ways of traversing the graph to get to the same node is the `shortestPath()` function. Using the shortest path between two nodes improves the performance of the query.

In this example, we want to discover a shortest path between the movies *The Matrix* and *A Few Good Men*. In our `MATCH` clause, we set the variable `p` to the result of calling `shortestPath()`, and then return `p`. In the call to `shortestPath()`, notice that we specify `*` for the relationship. This means any relationship; for the traversal.

```
MATCH p = shortestPath((m1:Movie)-[*]-(m2:Movie))
WHERE m1.title = 'A Few Good Men' AND
      m2.title = 'The Matrix'
RETURN p
```

```
[17]: Image('images/shortest_path.png', width=600)
```

```
[17]:
```



!!! When you use the `shortestPath()` function, the query editor will show a warning that this type of query could potentially run for a long time. You should heed the warning, especially for large graphs. When you use `ShortestPath()`, you can specify a upper limits for the shortest path. In addition, you should aim to provide the patterns for the from an to nodes that execute efficiently. For example, use labels and indexes.

**Specifying optional pattern matching** `OPTIONAL MATCH` matches patterns with your graph, just like `MATCH` does. The difference is that if no matches are found, `OPTIONAL MATCH` will use `NULLs` for missing parts of the pattern. `OPTIONAL MATCH` could be considered the Cypher equivalent of the outer join in SQL.

Here is an example where we query the graph for all people whose name starts with *James*. The `OPTIONAL MATCH` is specified to include people who have reviewed movies,

```

MATCH (p:Person)
WHERE p.name STARTS WITH 'James'
OPTIONAL MATCH (p)-[r:REVIEWED]->(m:Movie)
RETURN p.name, type(r), m.title

```

Notice that for all rows that do not have the `:REVIEWED` relationship, a null value is returned for the movie part of the query, as well as the relationship.

[18]: `Image('images/optional_match.png', width=600)`

[18]:



	p.name	type(r)	m.title
1	"James Marshall"	null	null
2	"James L. Brooks"	null	null
3	"James Cromwell"	null	null
4	"James Thompson"	"REVIEWED"	"The Replacements"
5	"James Thompson"	"REVIEWED"	"The Da Vinci Code"

Started streaming 5 records after 1 ms and completed after 5 ms.

**Aggregation in Cypher** Aggregation in Cypher is different from aggregation in SQL. In Cypher, you need not specify a grouping key. As soon as an aggregation function is used, all non-aggregated result columns become grouping keys. The grouping is implicitly done, based upon the fields in the `RETURN` clause.

**Collecting results** Cypher has a built-in function, `collect()` that enables you to aggregate a value into a list. Here is an example where we collect the list of movies that *Tom Cruise* acted in,

```

MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WHERE p.name = 'Tom Cruise'
RETURN collect(m.title) AS `movies for Tom Cruise`

```

In Cypher, there is no “GROUP BY” clause as there is in SQL. The graph engine uses non-aggregated columns as an automatic grouping key.

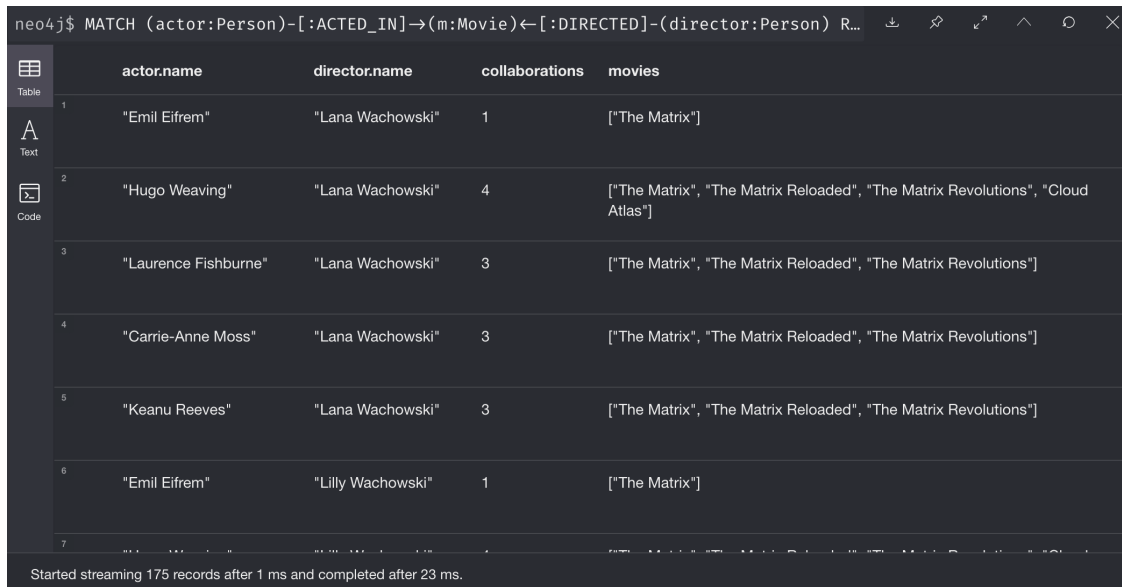
**Counting results** The Cypher `count()` function is very useful when you want to count the number of occurrences of a particular query result. If you specify `count(n)`, the graph engine calculates the number of occurrences of *n*. If you specify `count(*)`, the graph engine calculates the number of rows retrieved, including those with null values. When you use `count()`, the graph engine does an implicit group by based upon the aggregation.

Here is an example where we count the paths retrieved where an actor and director collaborated in a movie and the `count()` function is used to count the number of paths found for each actor/director collaboration.

```
MATCH (actor:Person)-[:ACTED_IN]->(m:Movie)<-[:DIRECTED]-(director:Person)
RETURN actor.name, director.name, count(m) AS collaborations, collect(m.title) AS movies
```

[19]: `Image('images/actor_director_collaborations.png', width=600)`

[19]:



The screenshot shows a Neo4j query result table with the following data:

	actor.name	director.name	collaborations	movies
1	"Emil Eifrem"	"Lana Wachowski"	1	["The Matrix"]
2	"Hugo Weaving"	"Lana Wachowski"	4	["The Matrix", "The Matrix Reloaded", "The Matrix Revolutions", "Cloud Atlas"]
3	"Laurence Fishburne"	"Lana Wachowski"	3	["The Matrix", "The Matrix Reloaded", "The Matrix Revolutions"]
4	"Carrie-Anne Moss"	"Lana Wachowski"	3	["The Matrix", "The Matrix Reloaded", "The Matrix Revolutions"]
5	"Keanu Reeves"	"Lana Wachowski"	3	["The Matrix", "The Matrix Reloaded", "The Matrix Revolutions"]
6	"Emil Eifrem"	"Lilly Wachowski"	1	["The Matrix"]
7				

Started streaming 175 records after 1 ms and completed after 23 ms.

**Additional processing using WITH** During the execution of a `MATCH` clause, you can specify that you want 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. You use the `WITH` clause to perform intermediate processing or data flow operations.

Here is an example where we start the query processing by retrieving all actors and their movies. During the query processing, want to only return actors that have 2 or 3 movies. All other actors and the aggregated results are filtered out. This type of query is a replacement for SQL's "HAVING" clause. The `WITH` clause does the counting and collecting, but is then used in the subsequent `WHERE` clause to limit how many paths are visited.

```
MATCH (a:Person)-[:ACTED_IN]->(m:Movie)
WITH a, count(a) AS numMovies, collect(m.title) as movies
WHERE numMovies > 1 AND numMovies < 4
RETURN a.name, numMovies, movies
```

[20]: `Image('images/actors_with_2or3_movies.png', width=600)`

[20]:



```
neo4j$ MATCH (a:Person)-[:ACTED_IN]->(m:Movie) WITH a, count(a) AS numMovies, colle...
```

	a.name	numMovies	movies
1	"Laurence Fishburne"	3	["The Matrix", "The Matrix Reloaded", "The Matrix Revolutions"]
2	"Carrie-Anne Moss"	3	["The Matrix", "The Matrix Reloaded", "The Matrix Revolutions"]
3	"Charlize Theron"	2	["The Devil's Advocate", "That Thing You Do"]
4	"J.T. Walsh"	2	["A Few Good Men", "Hoffa"]
5	"Kiefer Sutherland"	2	["A Few Good Men", "Stand By Me"]
6	"Kevin Bacon"	3	["A Few Good Men", "Frost/Nixon", "Apollo 13"]
7			

Started streaming 29 records after 1 ms and completed after 15 ms.

When you use the `WITH` clause, you specify the variables from the previous part of the query you want to pass on to the next part of the query. In this example, the variable `a` is specified to be passed on in the query, but `m` is not. Since `m` is not specified to be passed on, `m` will not be available later in the query. Notice that for the `RETURN` clause, `a`, `numMovies`, and `movies` are available for use.

Notice that, you have to name all expressions with an alias in a `WITH` that are not simple variables.

Here is another example where we want to find all actors who have acted in at least five movies, and find (optionally) the movies they directed and return the person and those movies.

```
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
```

In this example, we first retrieve all people, but then specify a pattern in the `WITH` clause where we calculate the number of `:ACTED_IN` relationships retrieved using the `size()` function. If this value is greater than five, we then also retrieve the `:DIRECTED` paths to return the name of the person and the title of the movie they directed. In the result, we see that these actors acted in more than five movies, but *Tom Hanks* is the only actor who directed a movie and thus the only person to have a value for the movie.

```
[21]: Image('images/popular_actors_with_at_least5_movies.png', width=600)
```

```
[21]:
```

```
neo4j$ MATCH (p:Person) WITH p, size((p)-[:ACTED_IN]->(:Movie)) AS movies WHERE movi...
```

	p.name	m.title
1	"Keanu Reeves"	null
2	"Hugo Weaving"	null
3	"Jack Nicholson"	null
4	"Meg Ryan"	null
5	"Tom Hanks"	"That Thing You Do"

Started streaming 5 records after 1 ms and completed after 17 ms.

**Controlling how results are returned** You have seen a number of query results where there is duplication in the results returned. In most cases, you want to eliminate duplicated results. You do so by using the `DISTINCT` keyword.

Here is a simple example where duplicate data is returned. *Tom Hanks* both acted in and directed the movie, *That Thing You Do*, so the movie is returned twice in the result stream,

```
MATCH (p:Person)-[:DIRECTED | :ACTED_IN]->(m:Movie)
WHERE p.name = 'Tom Hanks' AND m.released = 1996
RETURN m.released, collect(m.title) AS movies
```

We can eliminate the duplication by specifying the `DISTINCT` keyword as follows,

```
MATCH (p:Person)-[:DIRECTED | :ACTED_IN]->(m:Movie)
WHERE p.name = 'Tom Hanks'
RETURN m.released, collect(DISTINCT m.title) AS movies
```

```
[22]: x = Image('images/duplication.png', width=600)
      y = Image('images/no_duplication.png', width=600)

      display(x, y)
```

```
neo4j$ MATCH (p:Person)-[:DIRECTED | :ACTED_IN]->(m:Movie) WHERE p.name = 'Tom Hank...
```

	m.released	movies
1	1996	["That Thing You Do", "That Thing You Do"]

Started streaming 1 records after 1 ms and completed after 2 ms.

neo4j\$ MATCH (p:Person)-[:DIRECTED | :ACTED\_IN]->(m:Movie) WHERE p.name = 'Tom Hank...

	m.released	movies
1	1996	["That Thing You Do"]

Started streaming 1 records after 1 ms and completed after 2 ms.

### Using WITH and DISTINCT to eliminate duplication

```
MATCH (p:Person)-[:DIRECTED | :ACTED_IN]->(m:Movie)
WHERE p.name = 'Tom Hanks'
WITH DISTINCT m
RETURN m.released, m.title
```

**Ordering results** If you want the results to be sorted, you specify the expression to use for the sort using the **ORDER BY** keyword and whether you want the order to be descending using the **DESC** keyword. Ascending order is the default. Note that you can provide multiple sort expressions and the result will be sorted in that order. Just as you can use **DISTINCT** with **WITH** to eliminate duplication, you can use **ORDER BY** with **WITH** to control the sorting of results.

```
MATCH (p:Person)-[:DIRECTED | :ACTED_IN]->(m:Movie)
WHERE p.name = 'Tom Hanks'
RETURN m.released, collect(DISTINCT m.title) AS movies ORDER BY m.released DESC
```

**Limiting the number of results** Although you can filter queries to reduce the number of results returned, you may also want to limit the number of results. This is useful if you have very large result sets and you only need to see the beginning or end of a set of ordered results. You can use the **LIMIT** keyword to specify the number of results returned. Furthermore, you can use the **LIMIT** keyword with the **WITH** clause to limit results.

```
MATCH (m:Movie)
RETURN m.title as title, m.released as year ORDER BY m.released DESC LIMIT 10
```

### Controlling the number of results using WITH

```
MATCH (a:Person)-[:ACTED_IN]->(m:Movie)
WITH a, count(*) AS numMovies, collect(m.title) as movies
WHERE numMovies = 5
RETURN a.name, numMovies, movies
```

## Working with Cypher data

**Lists** There are many built-in Cypher functions that you can use to build or access elements in lists. A Cypher **map** is list of key/value pairs where each element of the list is of the format key: value. For example, a map of months and the number of days per month could be,

```
[Jan: 31, Feb: 28, Mar: 31, Apr: 30 , May: 31, Jun: 30 , Jul: 31, Aug: 31, Sep: 30, Oct: 31, Nov: 30, Dec: 31]
```

You can collect values for a list during a query and when you return results, you can sort by the size of the list using the `size()` function as follows,

```
MATCH (a:Person)-[:ACTED_IN]->(m:Movie)
WITH m, count(m) AS numCast, collect(a.name) as cast
RETURN m.title, cast, numCast ORDER BY size(cast)
```

**Unwinding lists** There may be some situations where you want to perform the opposite of collecting results, but rather separate the lists into separate rows. This functionality is done using the **UNWIND** clause.

Here is an example where we create a list with three elements, unwind the list and then return the values. Since there are three elements, three rows are returned with the values,

```
WITH [1, 2, 3] AS list
UNWIND list AS row
RETURN list, row
```

!!! Notice that there is no **MATCH** clause. You need not query the database to execute Cypher statements, but you do need the **RETURN** clause here to return the calculated values from the Cypher query.

!!! The **UNWIND** clause is frequently used when importing data into a graph.

**Dates** Cypher has a built-in `date()` function, as well as other temporal values and functions that you can use to calculate temporal values. You use a combination of numeric, temporal, spatial, list and string functions to calculate values that are useful to your application. For example, suppose you wanted to calculate the age of a Person node, given a year they were born (the `born` property must exist and have a value).

```
MATCH (actor:Person)-[:ACTED_IN]->(:Movie)
WHERE exists(actor.born)
WITH DISTINCT actor, date().year - actor.born as age
RETURN actor.name, age as `age today`
ORDER BY actor.born DESC
```

**QUIZ Q1)** Suppose you want to add a **WHERE** clause at the end of this statement to filter the results retrieved.

```
MATCH (p:Person)-[rel]->(m:Movie)<-[:PRODUCED]-(:Person)
```

What variables, can you test in the **WHERE** clause?

- p
- rel

- m
- ☐ PRODUCED

**Q2)** Suppose you want to retrieve all movies that have a released property value that is 2000, 2002, 2004, 2006, or 2008. Here is an incomplete Cypher example to return the title property values of all movies released in these years.

```
MATCH (m:Movie)
WHERE m.released XX [2000, 2002, 2004, 2006, 2008]
RETURN m.title
```

What keyword do you specify for XX?

- ☐ CONTAINS
- IN
- ☐ IS
- ☐ EQUALS

**Q3)** Given this Cypher query,

```
MATCH (a:Person)-[:ACTED_IN]->(m:Movie)
WITH m, count(m) AS numMovies, collect(m.title) as movies
WHERE numMovies > 1 AND numMovies < 4
RETURN //??
```

What variables or aliases can be used to return values?

- ☐ a
- m
  - numMovies
  - movies

```
[23]: from IPython.display import Image
```

### 0.0.5 Creating Nodes and Relationships

**Creating nodes** 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})
```

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 retrieve the node. Provided you have a reference to the node (for example, using a **MATCH** clause), you can always add, update, or remove labels and properties at a later time.

Here are some examples of creating a single node in Cypher,

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

Add a node with two labels to the graph of types `Movie` and `Action` with the title `Batman Begins`. This node can be retrieved using the title. A set of nodes with the labels `Movie` or `Action` can also be retrieved which will contain this node,

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

The variable `m` can be used for later processing after the `CREATE` clause in following query,

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

Also, you can see the example about the above situation,

```
CREATE` (m:Movie:Action {title: ' Batman Begins'})
RETURN m.title
```

!!! When the graph engine creates a node, it automatically assigns a read-only, unique ID to the node. This is not a property of a node, but rather an internal value.

**Creating multiple nodes** You can create multiple nodes by simply separating the nodes specified with commas, or by specifying multiple `CREATE` statements.

```
CREATE
(:Person {name: 'Michael Caine', born: 1933}),
(:Person {name: 'Liam Neeson', born: 1952}),
(:Person {name: 'Katie Holmes', born: 1978}),
(:Person {name: 'Benjamin Melniker', born: 1913})
```

## IMPORTANT

The graph engine will create a node with the same properties of a node that already exists. You can prevent this from happening in one of two ways,

- You can use `MERGE` rather than `CREATE` when creating the node.
- You can add constraints to your graph.

**Adding labels to a node** You may not know ahead of time what label or labels you want for a node when it is created. You can add labels to a node using the `SET` clause.

```
// adding one label to node referenced by the variable x
SET x:Label
```

```
// adding two labels to node referenced by the variable x
SET x:Label1:Label2
```

If you attempt to add a label to a node for which the label already exists, the `SET` processing is ignored.

!!! Notice here that we call the built-in function, `labels()` that returns the set of labels for the node.

## Removing labels from a node

```
// remove the label from the node referenced by the variable x
REMOVE x:Label
```

If you attempt to remove a label from a node for which the label does not exist, the **SET** processing is ignored.

```
MATCH (m:Movie:Action)
WHERE m.title = 'Batman Begins'
REMOVE m:Action
RETURN labels(m)
```

**Adding properties to a node** After you have created a node and have a reference to the node, you can add properties to the node, again using the **SET** keyword.

Here are simplified syntax examples for adding properties to a node referenced by the variable *x*,

```
SET x.propertyName = value
SET x.propertyName1 = value1, x.propertyName2 = value2
SET x = {propertyName1: value1, propertyName2: value2}
SET x += {propertyName1: value1, propertyName2: value2}
```

## IMPORTANT

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.

Note that the type of data for a property is not enforced. That is, you can assign a string value to a property that was once a numeric value and visa versa.

When you specify the JSON-style object for assignment (using **=**) of the property values for the node, the object must include all of the properties and their values for the node as the existing properties for the node are overwritten. However, if you specify **+=** when assigning to a property, the value at valueX is updated if the propertyNameX exists for the node. If the propertyNameX does not exist for the node, then the property is added to the node.

Here is an example where we 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
```

Here is another example where we set the property values to the movie node using the JSON-style object containing the property keys and values. Note that all properties must be included in the object.

```
MATCH (m:Movie)
WHERE m.title = 'Batman Begins'
SET m = {title: 'Batman Begins',
        released: 2005,
        lengthInMinutes: 140,
        videoFormat: 'DVD',
        grossMillions: 206.5}
```

```
RETURN m
```

Note that when you add a property to a node for the first time in the graph, the property key is added to the graph. So for example, in the previous example, we added the *videoFormat* and *grossMillions* property keys to the graph as they have never been used before for a node in the graph. Once a property key is added to the graph, it is never removed. When you examine the property keys in the database (by executing `CALL db.propertyKeys()`), you will see all property keys created for the graph, regardless of whether they are currently used for nodes and relationships.

Here is an example where we use the JSON-style object to add the *awards* property to the node and update the *grossMillions* property,

```
MATCH (m:Movie)
WHERE m.title = 'Batman Begins'
SET m += {grossMillions: 300,
         awards: 66}
RETURN m
```

**Removing properties from a node** There are two ways that you can remove a property from a node. One way is to use the `REMOVE` keyword. The other way is to set the property's value to `null`.

Here are simplified syntax examples for removing properties from a node referenced by the variable `x`,

```
REMOVE x.propertyName
SET x.propertyName = null
```

Suppose we determined that no other *Movie* node in the graph has the properties, *videoFormat* and *grossMillions*. There is no restriction that nodes of the same type must have the same properties. However, we have decided that we want to remove these properties from this node. Here is example Cypher to remove this property from this *Batman Begins* node,

```
MATCH (m:Movie)
WHERE m.title = 'Batman Begins'
SET m.grossMillions = null
REMOVE m.videoFormat
RETURN m
```

**Creating relationships** The connections capture the semantic relationships and context of the nodes in the graph.

Here is the simplified syntax for creating a relationship between two nodes referenced by the variables `x` and `y`,

```
CREATE (x)-[:REL_TYPE]->(y)
CREATE (x)<-[:REL_TYPE]-(y)
```

**When you create the relationship, it must have direction.** You can query nodes for a relationship in either direction, but you must create the relationship with a direction. An exception to this is when you create a node using `MERGE` that you will learn about later in this module.



In most cases, unless you are connecting nodes at creation time, you will retrieve the two nodes, each with their own variables, for example, by specifying a `WHERE` clause to find them, and then use the variables to connect them.

Here is an example. We want to connect the actor, *Michael Caine* with the movie, *Batman Begins*. We first retrieve the nodes of interest, then we create the relationship,

```
MATCH (a:Person), (m:Movie)
WHERE a.name = 'Michael Caine' AND m.title = 'Batman Begins'
CREATE (a)-[:ACTED_IN]->(m)
RETURN a, m
```

!!! Before you run these Cypher statements, you may see a warning in Neo4j Browser that you are creating a query that is a cartesian product that could potentially be a performance issue. You will see this warning if you have no unique constraint on the lookup keys. You will learn about uniqueness constraints later in the next module. If you are familiar with the data in the graph and can be sure that the `MATCH` clauses will not retrieve large amounts of data, you can continue. In our case, we are simply looking up a particular *Person* node and a particular *Movie* node so we can create the relationship.

You can create multiple relationships at once by simply providing the pattern for the creation that includes the relationship types, their directions, and the nodes that you want to connect.

Here is an example where we have already created *Person* nodes for an actor, *Liam Neeson*, and a producer, *Benjamin Melniker*. We create two relationships in this example, one for `ACTED_IN` and one for `PRODUCED`.

```
MATCH (a:Person), (m:Movie), (p:Person)
WHERE a.name = 'Liam Neeson' AND
      m.title = 'Batman Begins' AND
      p.name = 'Benjamin Melniker'
CREATE (a)-[:ACTED_IN]->(m)<-[:PRODUCED]-(p)
RETURN a, m, p
```

!!! When you create relationships based upon a `MATCH` clause, you must be certain that only a single node is returned for the `MATCH`, otherwise multiple relationships will be created.

**Adding properties to relationships** All the things described for adding properties to nodes also apply to relationships. You can look the **Adding properties to a node** section. Here are the examples,

```
MATCH (a:Person), (m:Movie)
WHERE a.name = 'Christian Bale' AND m.title = 'Batman Begins'
CREATE (a)-[rel:ACTED_IN]->(m)
SET rel.roles = ['Bruce Wayne','Batman']
RETURN a, m

MATCH (a:Person), (m:Movie)
WHERE a.name = 'Christian Bale' AND m.title = 'Batman Begins'
CREATE (a)-[:ACTED_IN {roles: ['Bruce Wayne', 'Batman']}]>(m)
RETURN a, m
```

By default, the graph engine will create a relationship between two nodes, even if one already exists. You can test to see if the relationship exists before you create it as follows,

```
MATCH (a:Person),(m:Movie)
WHERE a.name = 'Christian Bale' AND
      m.title = 'Batman Begins' AND
      NOT exists((a)-[:ACTED_IN]->(m))
CREATE (a)-[rel:ACTED_IN]->(m)
SET rel.roles = ['Bruce Wayne','Batman']
RETURN a, rel, m
```

!!! You can prevent duplication of relationships by merging data using the **MERGE** clause, rather than the **CREATE** clause. You will learn about merging data later in this module.

**Removing properties from a relationship** All the things described for removing properties to nodes also apply to relationships. You can look the **Removing properties from a node** section. Here are the examples,

```
MATCH (a:Person)-[rel:ACTED_IN]->(m:Movie)
WHERE a.name = 'Christian Bale' AND m.title = 'Batman Begins'
REMOVE rel.roles
RETURN a, rel, m
```

**Deleting nodes and relationships** If a node has no relationships to any other nodes, you can simply delete it from the graph using the **DELETE** clause. Relationships are also deleted using the **DELETE** clause.

!!! If you attempt to delete a node in the graph that has relationships in or out of the node, the graph engine will return an error because deleting such a node will leave orphaned relationships in the graph.

**Deleting relationships** You can delete a relationship between nodes by first finding it in the graph and then deleting it.

In this example, we want to delete the *ACTED\_IN* relationship between *Christian Bale* and the movie *Batman Begins*. We find the relationship, and then delete it,

```
MATCH (a:Person)-[rel:ACTED_IN]->(m:Movie)
WHERE a.name = 'Christian Bale' AND m.title = 'Batman Begins'
DELETE rel
RETURN a, m
```

In this example, we find the node for the producer, Benjamin Melniker, as well as his relationship to movie nodes. First, we delete the relationship(s), then we delete the node,

```
MATCH (p:Person)-[rel:PRODUCED]->(m:Movie)
WHERE p.name = 'Benjamin Melniker'
DELETE rel, p
```

**Deleting nodes and relationships** The most efficient way to delete a node and its corresponding relationships is to specify `DETACH DELETE`. When you specify `DETACH DELETE` for a node, the relationships to and from the node are deleted, then the node is deleted.

If we were to attempt to delete the *Keanu Reeves* node without first deleting its relationships,

```
MATCH (p:Person)
WHERE p.name = 'Keanu Reeves'
DELETE p
```

We would see error like that,

[24]: `Image('images/keanu_reeves_delete_error.png', width=600)`

[24]:



Here we delete the *Keanu Reeves* node and its relationships to any other nodes,

```
MATCH (p:Person)
WHERE p.name = 'Keanu Reeves'
DETACH DELETE p
```

**Merging data in the graph** You can use `MERGE` to either create new nodes and relationships or to make structural changes to existing nodes and relationships.

[25]: `Image('images/create_statement.png')`

[25]:

If you use <code>CREATE</code> ,	The result is,
Node	If a node with the same property values 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. If you specify a set of properties to be created using <code>=</code> rather than <code>+=</code> , existing properties are removed if they are not included in the set.
Relationship	If the relationship exists, a duplicate relationship is created.

!!! You should never create duplicate nodes or relationships in a graph.

The `MERGE` clause is used to find elements in the graph. If the element is not found, it is created.

You use the `MERGE` clause to,

- 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.

**Using MERGE to create nodes** Here is the simplified syntax for the MERGE clause for creating a node,

```
MERGE (variable:Label{nodeProperties})
RETURN variable
```

If there is an existing node with *Label* and *nodeProperties* found in the graph, no node is created. If, however the node is not found in the graph, then the node is created.

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

**Using MERGE to create relationships** Here is the syntax for the MERGE clause for creating relationships,

```
MERGE (variable:Label {nodeProperties})-[:REL_TYPE]->(otherNode)
RETURN variable
```

If there is an existing node with *Label* and *nodeProperties* with the *:REL\_TYPE* to *otherNode* found in the graph, no relationship is created. If the relationship does not exist, it is created.

Although, you can leave out the direction of the relationship being created with the MERGE, in which case a left-to-right arrow will be assumed, a best practice is to always specify the direction of the relationship. However, if you have bidirectional relationships and you want to avoid creating duplicate relationships, you must leave off the arrow.

**Specifying creation behavior when merging** You can use the MERGE clause, along with ON CREATE to assign specific values to a node being created as a result of an attempt to merge.

```
MERGE (a:Person {name: 'Sir Michael Caine'})
ON CREATE SET a.birthPlace = 'London', a.born = 1934
RETURN a
```

We know that there are no existing *Sir Michael Caine* Person nodes. When the MERGE executes, it will not find any matching nodes so it will create one and will execute the ON CREATE clause where we set the birthplace and born property values.

You can also specify an ON MATCH clause during merge processing. If the exact node is found, you can update its properties or labels. Here is an 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
```

**Using MERGE to create relationships** Using MERGE to create relationships is expensive and you should only do it when you need to ensure that a relationship is unique and you are not sure if it already exists.

In this example, we use the MATCH clause to find all *Person* nodes that represent *Michael Caine* and we find the movie, *Batman Begins* that we want to connect to all of these nodes. We already have a connection between one of the *Person* nodes and the *Movie* node. We do not want this relationship to be duplicated. This is where we can use MERGE as follows,

```
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. MERGE tries to find a full pattern and if it doesn't find it, it creates that full pattern. That's why in most cases you should first MERGE your nodes and then your relationship afterwards.

Only if you intentionally want to create a node within the context of another (like a month within a year) then a MERGE pattern with one bound and one unbound node makes sense.

```
MERGE (fromDate:Date {year: 2018})<-[:IN_YEAR]-(toDate:Date {month: 'January'})
```

**QUIZ Q1)** What Cypher clauses can you use to create a node?

- CREATE
- ☐ CREATE NODE
- MERGE
- ☐ ADD

**Q2)** Suppose that you have retrieved a node, s with a property, *color*:

```
MATCH (s:Shape {location: [20,30]})
???
RETURN s
```

What Cypher clause do you add here to delete the color property from this node?

- ☐ DELETE s.color
- SET s.color=null
- REMOVE s.color
- ☐ SET s.color=?

**Q3)** Suppose you retrieve a node, n in the graph that is related to other nodes. What Cypher clause do you write to delete this node and its relationships in the graph?

- ☐ DELETE n
- ☐ DELETE n WITH RELATIONSHIPS
- ☐ REMOVE n
- DETACH DELETE n

```
[26]: from IPython.display import Image
```

### 0.0.6 Getting More Out of Neo4j

**Using Cypher parameters** In your Cypher statements, a parameter name begins with the \$ symbol.

Here is an example where we have parameterized the query,

```
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WHERE p.name = $actorName
RETURN m.released, m.title ORDER BY m.released DESC
```

At runtime, if the parameter `$actorName` has a value, it will be used in the Cypher statement when it runs in the graph engine.

In Neo4j Browser, you can set values for Cypher parameters that will be in effect during your session.

You can set the value of a single parameter in the query editor pane as shown in this example where the value Tom Hanks is set for the parameter `actorName`,

```
:param actorName => 'Tom Hanks'
```

Notice here that `:param` is a client-side browser command. It takes a name and expression and stores the value of that expression for the name in the session.

You can also use the JSON-style syntax to set all of the parameters in your Neo4j Browser session. The values you can specify in this object are numbers, strings, and booleans. In this example we set two parameters for our session,

```
:params {actorName: 'Tom Cruise', movieName: 'Top Gun'}
```

If you want to remove an existing parameter from your session, you do so by using the JSON-style syntax and excluding the parameter for your session.

If you want to view the current parameters and their values, simply type `:params:`

If you want to clear all parameters, you can simply type:

```
:params {}
```

**Analyzing Cypher execution** There are two Cypher keywords you can prefix a Cypher statement with to analyze a query,

- **EXPLAIN** provides estimates of the graph engine processing that will occur, but does not execute the Cypher statement.
- **PROFILE** provides real profiling information for what has occurred in the graph engine during the query and executes the Cypher statement.
- The **EXPLAIN** option provides the Cypher query plan. You can compare different Cypher statements to understand the stages of processing that will occur when the Cypher executes.

Here is an example where we have set the `actorName` and `year` parameters for our session and we execute this Cypher statement,

```
EXPLAIN MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WHERE p.name = $actorName AND
```

```
m.released < $year
RETURN p.name, m.title, m.released
```

You can expand each phase of the Cypher execution to examine what code is expected to run. Each phase of the query presents you with an estimate of the number of rows expected to be returned. With **EXPLAIN**, the query does not run, the graph engine simply produces the query plan.

For a better metric for analyzing how the Cypher statement will run you use the **PROFILE** keyword which runs the Cypher statement and gives you run-time performance metrics.

In the result of **PROFILE**, we can see that for each phase of the graph engine processing, we can view the cache hits and most importantly the number of times the graph engine accessed the database (db hits). This is an important metric that will affect the performance of the Cypher statement at run-time.

**Monitoring queries** If you are testing an application and have run several queries against the graph, there may be times when your Neo4j Browser session hangs with what seems to be a very long-running query. There are two reasons why a Cypher query may take a long time,

- The query returns a lot of data. The query completes execution in the graph engine, but it takes a long time to create the result stream.
  - Example: `MATCH (a)--(b)--(c)--(d)--(e)--(f) RETURN a`
- The query takes a long time to execute in the graph engine.
  - Example: `MATCH (a), (b), (c), (d), (e) RETURN count(id(a))`

If the query executes and then returns a lot of data, there is no way to monitor it or kill the query. All that you can do is close your Neo4j Browser session and start a new one. If the server has many of these rogue queries running, it will slow down considerably so you should aim to limit these types of queries. If you are running Neo4j Desktop, you can simply restart the database to clear things up, but if you are using a Neo4j Sandbox, you cannot do so. The database server is always running and you cannot restart it. Your only option is to shut down the Neo4j Sandbox and create a new Neo4j Sandbox, but then you lose any data you have worked with.

If, however, the query is a long-running query, you can monitor it by using the **:queries** command.

The **:queries** command calls `dbms.listQueries` under the hood, which is why we see two queries here. We have turned on **AUTO-REFRESH** so we can monitor the number of ms used by the graph engine thus far. You can kill the running query by double-clicking the icon in the Kill column. Alternatively, you can execute the statement `CALL dbms.killQuery('query-id')`.

!!! Please notice that, these functions are only available in Enterprise Edition of Neo4j.

**Managing constraints and node keys** You have seen that you can accidentally create duplicate nodes in the graph if you're not protected. In most graphs, you will want to prevent duplication of data. Unfortunately, you cannot prevent duplication by checking the existence of the exact node (with properties) as this type of test is not cluster or multi-thread safe as no locks are used. This is one reason why **MERGE** is preferred over **CREATE**, because **MERGE** does use locks.

In addition, you have learned that a node or relationship need not have a particular property. What if you want to ensure that all nodes or relationships of a specific type (label) must set values for certain properties?

A third scenario with graph data is where you want to ensure that a set of property values for nodes of the same type, have a unique value. This is the same thing as a primary key in a relational database.

All of these scenarios are common in many graphs. In Neo4j, you can use Cypher to,

- Add a uniqueness constraint that ensures that a value for a property is unique for all nodes of that type.
- Add an existence constraint that ensures that when a node or relationship is created or modified, it must have certain properties set.
- Add a node key that ensures that a set of values for properties of a node of a given type is unique.

!!! Please notice that, existence constraints and node keys are only available in Enterprise Edition of Neo4j.

**Ensuring that a property value for a node is unique** You add a uniqueness constraint to the graph by creating a constraint that asserts that a particular node property is unique in the graph for a particular type of node.

Here is an example for ensuring that the title for a node of type *Movie* is unique,

```
CREATE CONSTRAINT ON (m:Movie) ASSERT m.title IS UNIQUE
```

This Cypher statement will fail if the graph already has multiple *Movie* nodes with the same value for the *title* property. Note that you can create a uniqueness constraint, even if some *Movie* nodes do not have a *title* property.

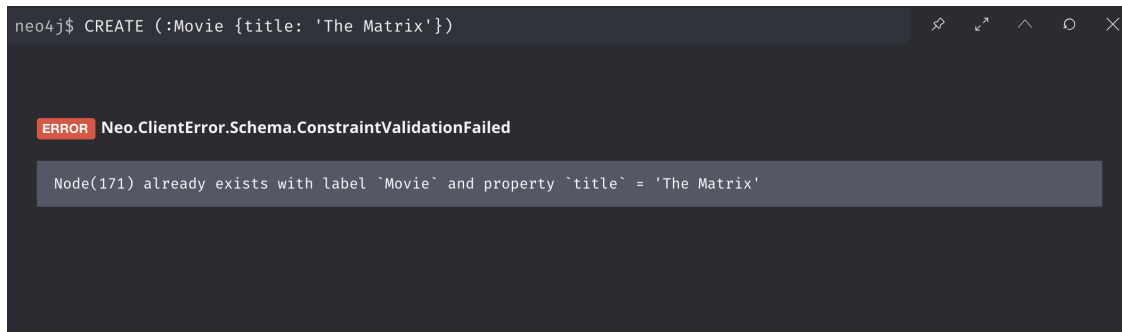
And if we attempt to create a *Movie* with the *title*, *The Matrix*, the Cypher statement will fail because the graph already has a movie with that title,

```
CREATE (:Movie {title: 'The Matrix'})
```

Here is the result of running this Cypher statement on the *Movie* graph,

[27]:  Image('images/uniqueness\_failed.png', width=600)

[27]:



**Ensuring that properties exist** Having uniqueness for a property value is only useful in the graph if the property exists. In most cases, you will want your graph to also enforce the existence



of properties, not only for those node properties that require uniqueness, but for other nodes and relationships where you require a property to be set. Uniqueness constraints can only be created for nodes, but existence constraints can be created for node or relationship properties.

You add an existence constraint to the graph by creating a constraint that asserts that a particular type of node or relationship property must exist in the graph when a node or relationship of that type is created or updated.

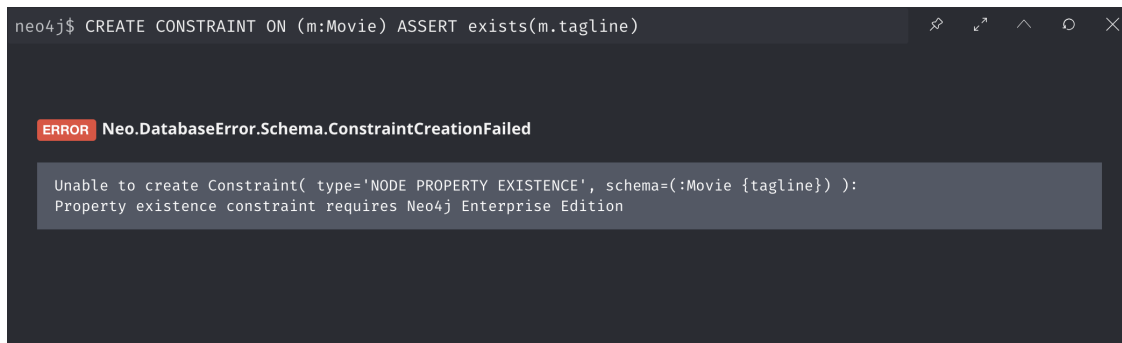
Here is an example for adding the existence constraint to the *tagline* property of all *Movie* nodes in the graph,

```
CREATE CONSTRAINT ON (m:Movie) ASSERT exists(m.tagline)
```

Here is the result of running this Cypher statement,

[28]: `Image('images/constraint_exist_tagline_failure.png', width=600)`

[28]:



The constraint cannot be added to the graph because a node has been detected that violates the constraint.

We know that in the *Movie* graph, all *:REVIEWED* relationships currently have a property, *rating*. We can create an existence constraint on that property as follows,

```
CREATE CONSTRAINT ON ()-[rel:REVIEWED]-() ASSERT exists(rel.rating)
```

Notice that when you create the constraint on a relationship, you need not specify the direction of the relationship. Also, we need to remember, property existence constraint only available in Enterprise Edition of Neo4j.

**Retrieving constraints defined for the graph** You can run the browser command `:schema` to view existing indexes and constraints defined for the graph.

Just as you have used other *db* related methods to query the schema of the graph, you can query for the set of constraints defined in the graph as follows,

```
CALL db.constraints()
```

Using the method notation for the CALL statement enables you to use the call for returning results that may be used later in the Cypher statement.

**Dropping constraints** You use similar syntax to drop an existence or uniqueness constraint, except that you use the `DROP` keyword rather than `CREATE`

Here we drop the existence constraint for the *rating* property for all *REVIEWED* relationships in the graph,

```
DROP CONSTRAINT ON ()-[rel:REVIEWED]-() ASSERT exists(rel.rating)
```

**Creating node keys** A node key is used to define the uniqueness constraint for multiple properties of a node of a certain type. A node key is also used as a composite index in the graph.

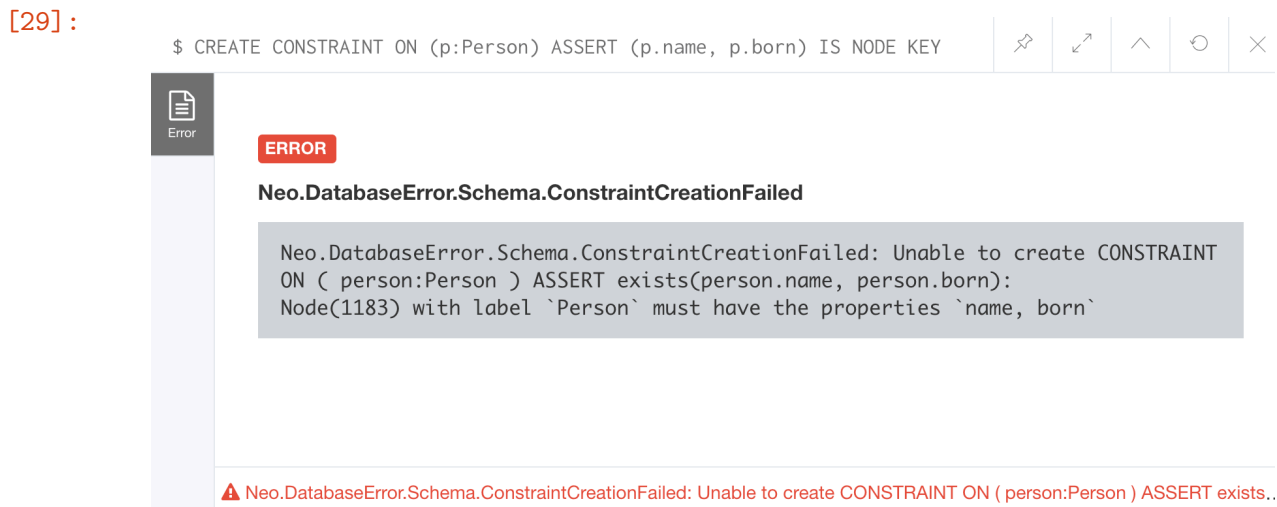
Suppose that in our *Movie* graph, we will not allow a *Person* node to be created where both the *name* and *born* properties are the same. We can create a constraint that will be a node key to ensure that this uniqueness for the set of properties is asserted.

Here is an example to create this node key,

```
CREATE CONSTRAINT ON (p:Person) ASSERT (p.name, p.born) IS NODE KEY
```

Here is the result of running this Cypher statement on our *Movie* graph,

[29]: 



This attempt to create the constraint failed because there are *Person* nodes in the graph that do not have the *born* property defined.

If we set these properties for all nodes in the graph that do not have *born* properties with,

```
MATCH (p:Person)
WHERE NOT exists(p.born)
SET p.born = 0
```

Then the creation of the node key succeeds.

Any subsequent attempt to create or modify an existing *Person* node with *name* or *born* values that violate the uniqueness constraint as a node key will fail.

**Managing indexes** The uniqueness and node key constraints that you add to a graph 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. You can have multiple properties on nodes that must be unique.

Here is a brief summary of when single-property indexes are 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.

## IMPORTANT

Because index maintenance incurs additional overhead when nodes are created, we recommend that for large graphs, indexes are created after the data has been loaded into the graph. You can view the progress of the creation of an index when you use the `:schema` command.

**Creating indexes** You create an index to improve graph engine performance. A unique constraint on a property is an index so you need not create an index for any properties you have created uniqueness constraints for. An index on its own does not guarantee uniqueness.

Here is an example of how we would create a single-property index on the *released* property of all nodes of type *Movie*:

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

- If a set of properties for a node must be unique for every node, then you should create a constraint as a node key, rather than an index.
- If, however, there can be duplication for a set of property values, but you want faster access to them, then you can create a composite index. A composite index is based upon multiple properties for a node.

!!! In general, you need not end a Cypher statement with a semi-colon, but if you want to execute multiple Cypher statements, you must separate them. You have already used the semi-colon to separate Cypher statements when you loaded the Movie database in the training exercises.

Now that the graph has *Movie* nodes with both the properties, *released* and *videoFormat*, we can create a composite index on these properties as follows,

```
CREATE INDEX ON :Movie(released, videoFormat)
```

**Retrieving indexes** Just as you can retrieve the constraints defined for the graph using `:schema` or `CALL db.constraints()`, you can retrieve the indexes,

```
CALL db.indexes()
```

!!! Notice that the unique constraints and node keys are also shown as indexes in the graph.

**Dropping indexes** Here is an example of dropping the composite index that we just created,

```
DROP INDEX ON :Movie(released, videoFormat)
```

**Importing data** In this module, you will be introduced to some simple steps for loading CSV data into your graph with Cypher. If you are interested in direct loading of data from a relational DBMS into a graph, you should read about the [Neo4j Extract Transform Load \(ETL\)](#) tool at , as well as many of the useful pre-written procedures that are available for your use in the APOC library.

In Cypher, you can,

- Load data from a URL (http(s) or file).
- Process data as a stream of records.
- Create or update the graph with the data being loaded.
- Use transactions during the load.
- Transform and convert values from the load stream.
- Load up to 10M nodes and relationships.

CSV import is commonly used to import data into a graph. If you want to import data from CSV, you will need to first develop a model that describes how data from your CSV maps to data in your graph.

**Importing normalized data using LOAD CSV** The `LOAD CSV` clause parses a local file in the import directory of your Neo4j installation or a remote file into a stream of rows which represent maps (with headers) or lists. Then you can use whichever Cypher operations you want to either create nodes or relationships or to merge with the existing graph.

Here is the simplified syntax for using `LOAD CSV`,

```
// row is a variable that is used to extract data
LOAD CSV WITH HEADERS FROM url-value
AS row
```

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 your system. Each line contains data that is interpreted as values for each column name. When each line is read from the file, you can perform the necessary processing to create or merge data into the graph.

As CSV files usually represent either node or relationship lists, you will run multiple passes to create nodes and relationships separately.

The `movies_to_load.csv` file (sample below) contains the data that will add *Movie* nodes,

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

Before you load data from CSV files into your graph, you should first confirm that the data retrieved looks OK. Rather than creating nodes or relationships, you can simply return information about the data to be loaded.

For example you can execute this Cypher statement to get a count of the data to be loaded from the **movies\_to\_load.csv** file so you have an idea of how much data will be loaded,

```
LOAD CSV WITH HEADERS
FROM 'http://data.neo4j.com/intro-neo4j/movies_to_load.csv'
AS line
RETURN count(*)
```

You might even want to visually inspect the data before you load it to see if it is what you were expecting,

```
LOAD CSV WITH HEADERS
FROM 'https://data.neo4j.com/intro-neo4j/movies_to_load.csv'
AS line
RETURN * LIMIT 1
```

Notice here that the *summary* column's data has an extra space before the data in the file. In order to ensure that all *tagline* values in our graph do not have an extra space, we will trim the value before assigning it to the tagline property. Once we are sure you want to load the data into your graph, we do so by assigning values from each row read in to a new node.

You may want to format the data before it is loaded to confirm it matches what you want in your graph,

```
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 following query creates the *Movie* nodes using some of the data from **movies\_to\_load.csv** as properties,

```
LOAD CSV WITH HEADERS
FROM 'https://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) })
```

We assign a value to *movieId* from the id data in the CSV file. In addition, we assign the data from *summary* to the *tagline* property, with a trim. We also convert the data read from *year* to an integer using the built-in function `toInteger()` before assigning it to the released property.

The **persons\_to\_load.csv** file (sample below) holds the data that will populate the *Person* nodes,

```
Id,name,birthyear
1,Charlie Sheen, 1965
2,Oliver Stone, 1946
3,Michael Douglas, 1944
4,Martin Sheen, 1940
5,Morgan Freeman, 1937
```

In case you already have people in your database, you will want to avoid creating duplicates. That's why instead of just creating them, we use `MERGE` to ensure unique entries after the import. We use the `ON CREATE` clause to set the values for *name* and *born*.

```
LOAD CSV WITH HEADERS
FROM 'https://data.neo4j.com/intro-neo4j/persons_to_load.csv'
AS line
MERGE (actor:Person { personId: line.Id })
ON CREATE SET actor.name = line.name,
             actor.born = toInteger(trim(line.birthyear))
```

There are a couple of things to note here. The name of the column is case-sensitive. In addition, notice that the data for the *birthyear* column has an extra space before the data. To allow this data to be converted to an integer, we must first trim the whitespace using the `trim()` built-in function.

The **roles\_to\_load.csv** file (sample below) holds the data that will populate the relationships between the nodes.

```
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 query below matches the entries of *line.personId* and *line.movieId* to their respective *Movie* and *Person* nodes, and creates an *ACTED\_IN* relationship between the person and the movie. This model includes a relationship property of *role*, which is passed via *line.role*.

```
LOAD CSV WITH HEADERS
FROM 'https://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)
```

**Importing denormalized data** If your file contains denormalized data, you can run the same file with multiple passes and simple operations as shown above. Alternatively, you might have to use `MERGE` to create nodes and relationships uniquely.

For our use case, we can import the data using a CSV structure like this,

```
title;released;summary;actor;birthyear;characters
Back to the Future;1985;17 year old Marty McFly got home early last night. 30 years early.;Michael J. Fox;1961;
Back to the Future;1985;17 year old Marty McFly got home early last night. 30 years early.;Chris Penn;1969;
```

Here are the Cypher statements to load this data,

```
LOAD CSV WITH HEADERS
FROM 'https://data.neo4j.com/intro-neo4j/movie_actor_roles_to_load.csv'
AS line FIELDTERMINATOR ';' ;
```

```

MERGE (movie:Movie { title: line.title })
ON CREATE SET movie.released = toInteger(line.released),
              movie.tagline = line.summary
MERGE (actor:Person { name: line.actor })
ON CREATE SET actor.born = toInteger(line.birthyear)
MERGE (actor)-[r:ACTED_IN]->(movie)
ON CREATE SET r.roles = split(line.characters,',')

```

Notice a couple of things in this Cypher statement. This file uses a semi-colon as a field terminator, rather than the default comma. In addition, the built-in method `split()` is used to create the list for the roles property.

**Importing a large dataset** If you import a larger amount of data (more than 10,000 rows), it is recommended to prefix your `LOAD CSV` clause with a `PERIODIC COMMIT` hint. This allows the database to regularly commit the import transactions to avoid memory churn for large transaction-states.

### 0.0.7 Exercises

**Exercise-1** Retrieve all `REVIEWED` relationships from the graph where the summary of the review contains the string `fun`, returning the movie title reviewed and the rating and summary of the relationship.

```

MATCH (:Person)-[r:REVIEWED]->(m:Movie)
WHERE toLower(r.summary) CONTAINS 'fun'
RETURN m.title as Movie, r.summary as Review, r.rating as Rating

```

**Exercise-2** Retrieve all people who have produced a movie, but have not directed a movie, returning their names and the movie titles.

```

MATCH (a:Person)-[:PRODUCED]->(m:Movie)
WHERE NOT ((a)-[:DIRECTED]->(:Movie))
RETURN a.name, m.title

```

**Exercise-3** Retrieve the movies and their actors where one of the actors also directed the movie, returning the actors names, the director's name, and the movie title.

```

MATCH (a1:Person)-[:ACTED_IN]->(m:Movie)<-[:ACTED_IN]-(a2:Person)
WHERE exists((a2)-[:DIRECTED]->(m))
RETURN a1.name as Actor, a2.name as `Actor/Director`, m.title as Movie

```

**Exercise-4** 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 (a1:Person)-[:ACTED_IN]->(m:Movie)<-[:ACTED_IN]-(a2:Person)
WHERE exists((a2)-[:DIRECTED]->(m))
RETURN a1.name as Actor, a2.name as `Actor/Director`, m.title as Movie

```

**Exercise-5** Retrieve all movies that *Tom Cruise* has acted in and the co-actors that acted in the same movie, returning the movie title and the list of co-actors that *Tom Cruise* worked with.

```
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)<-[:ACTED_IN]-(p2:Person)
WHERE p.name = 'Tom Cruise'
RETURN m.title as movie, collect(p2.name) AS `co-actors`
```

**Exercise-6** Retrieve the actors who have acted in exactly five movies, returning the name of the actor, and the list of movies for that actor.

```
MATCH (a:Person)-[:ACTED_IN]->(m:Movie)
WITH a, count(m) AS numMovies, collect(m.title) AS movies
WHERE numMovies = 5
RETURN a.name, movies
```

**Exercise-7** Write a Cypher query that retrieves all actors that acted in movies, and also retrieves the producers for those movies. During the query, collect the names of the actors and the names of the producers. Return the movie titles, along with the list of actors for each movie, and the list of producers for each movie making sure there is no duplication of data. Order the results returned based upon the size of the list of actors.

```
MATCH (a:Person)-[:ACTED_IN]->(m:Movie),
      (m)<-[:PRODUCED]-(p:Person)
WITH m, collect(DISTINCT a.name) AS cast, collect(DISTINCT p.name) AS producers
RETURN DISTINCT m.title, cast, producers
ORDER BY size(cast)
```

**Exercise-8** Write a Cypher query that retrieves all actors that acted in movies, and collects the list of movies for any actor that acted in more than five movies. Return the name of the actor and the list.

```
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WITH p, collect(m) AS movies
WHERE size(movies) > 5
RETURN p.name, movies
```

Modify the query you just wrote so that before the query processing ends, you unwind the list of movies and then return the name of the actor and the title of the associated movie.

```
MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WITH p, collect(m) AS movies
WHERE size(movies) > 5
WITH p, movies UNWIND movies AS movie
RETURN p.name, movie.title
```

**Exercise-9** Add the label *OlderMovie* to any Movie node that was released before 2010.

```
MATCH (m:Movie)
WHERE m.released < 2010
SET m:OlderMovie
RETURN DISTINCT labels(m)
```



**Exercise-10** Create the *ACTED\_IN* relationship between the actors, *Robin Wright*, *Tom Hanks*, and *Gary Sinise* and the movie, *Forrest Gump*.

```
MATCH (p:Person), (m:Movie)
WHERE p.name IN ['Tom Hanks','Gary Sinise', 'Robin Wright']
      AND m.title = 'Forrest Gump'
MERGE (p)-[:ACTED_IN]->(m)
```

```
MATCH (p:Person)-[rel:ACTED_IN]->(m:Movie)
WHERE m.title = 'Forrest Gump'
SET rel.roles =
CASE p.name
  WHEN 'Tom Hanks' THEN ['Forrest Gump']
  WHEN 'Robin Wright' THEN ['Jenny Curran']
  WHEN 'Gary Sinise' THEN ['Lieutenant Dan Taylor']
END
```

```
MERGE (m:Movie {title: 'Forrest Gump'})
ON CREATE SET m.released = 1994
RETURN m
```

```
MERGE (m:Movie {title: 'Forrest Gump'})
ON CREATE SET m.released = 1994
ON MATCH SET m.tagline = "Life is like a box of chocolates...you never know what you're gonna g
RETURN m
```

```
:param year => 2000
```

date of the movie, the rating given to the movie by the reviewer, and the list of actors for that particular movie. Also use the *year* parameter in this query.

```
MATCH (r:Person)-[rel:REVIEWED]->(m:Movie)<-[:ACTED_IN]-(a:Person)
WHERE m.released = $year
RETURN DISTINCT r.name, m.title, m.released, rel.rating, collect(a.name)
```

Add a parameter named *ratingValue* to your session with a value of 65. Also, change the *year* parameter from 2000 to 2006.

```
:params {year: 2006, ratingValue: 65}
```

Modify the query you wrote previously to also filter the result returned by the *rating* for the movie.

```
MATCH (r:Person)-[rel:REVIEWED]->(m:Movie)<-[:ACTED_IN]-(a:Person)
WHERE m.released = $year AND
      rel.rating > $ratingValue
RETURN DISTINCT r.name, m.title, m.released, rel.rating, collect(a.name)
```