Introduction to Graph Databases

The evolution of graph databases

Today's business and user requirements demand applications that connect more and more of the world's data, yet still expect high levels of performance and data reliability. Many applications of the future will be built using graph databases like Neo4j.

In this video, you will learn how the need for graph databases has evolved.

What Is a graph database?

A graph database is an online database management system with Create, Read, Update and Delete (CRUD) operations working on a graph data model. Graph databases are generally built for use with online transaction processing (OLTP) systems. Accordingly, they are normally optimized for transactional performance, and engineered with transactional integrity and operational availability in mind.

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.

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

The case for graph databases

The biggest value that graphs bring to the development stack is their ability to store relationships and connections as first-class entities.

For instance, the early adopters of graph technology reimagined their businesses around the value of data relationships. These companies have now become industry leaders: LinkedIn, Google, Facebook and PayPal.

As pioneers in graph technology, each of these enterprises had to build their own graph database from scratch. Fortunately for today's developers, that's no longer the case, as graph database technology is now available off the shelf.

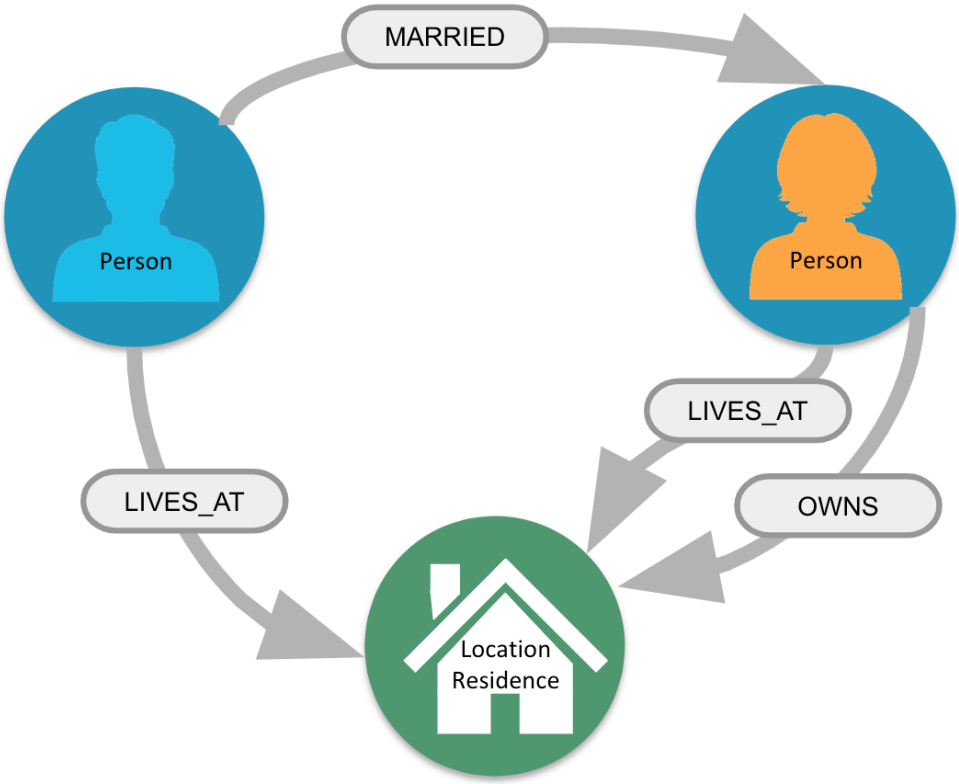
In this video, you will learn how graph databases help you to model real-world data that needs to be connected as well as how Neo4j is used to solve real problems facing enterprises today.

What is a graph?

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

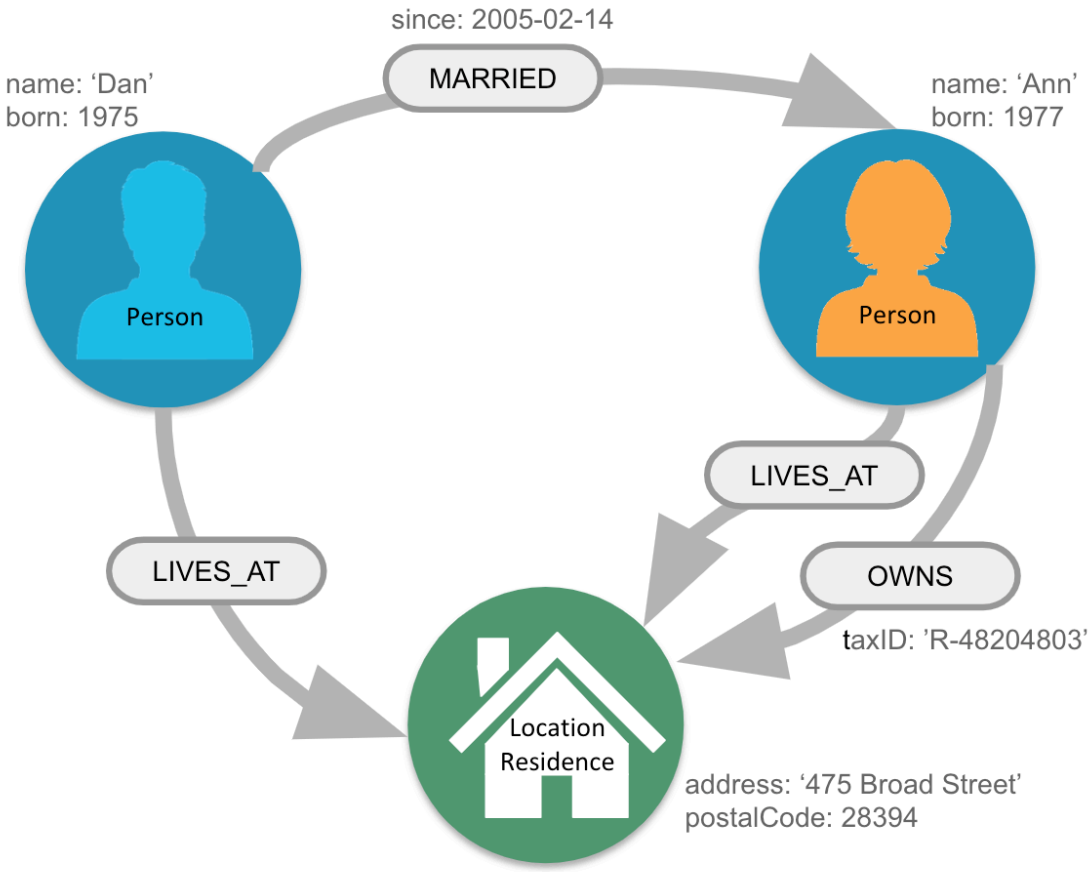
Each node represents an entity (a person, place, thing, category or other piece of data). With Neo4j, nodes can have **labels** that are used to define types for nodes. For example, a *Location* node is a node with the label *Location*. That same node can also have a label, *Residence*. Another *Location*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

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.



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.



This enables you to closely align data and connections in the graph to your real-world application. For example, a *Person* node might have a property, *name* and a *Location* node might have a property, *address*. In addition, a relationship, *MARRIED* , might have a property, *since*.

In this video, you will learn how to model property graphs containing nodes and relationships and how Cypher is used to access a graph database.

Modeling relational to graph

Many applications' data is modeled as relational data. There are some similarities between a relational model and a graph model:

Relational	Graph
Rows	Nodes
Joins	Relationships
Table names	Labels

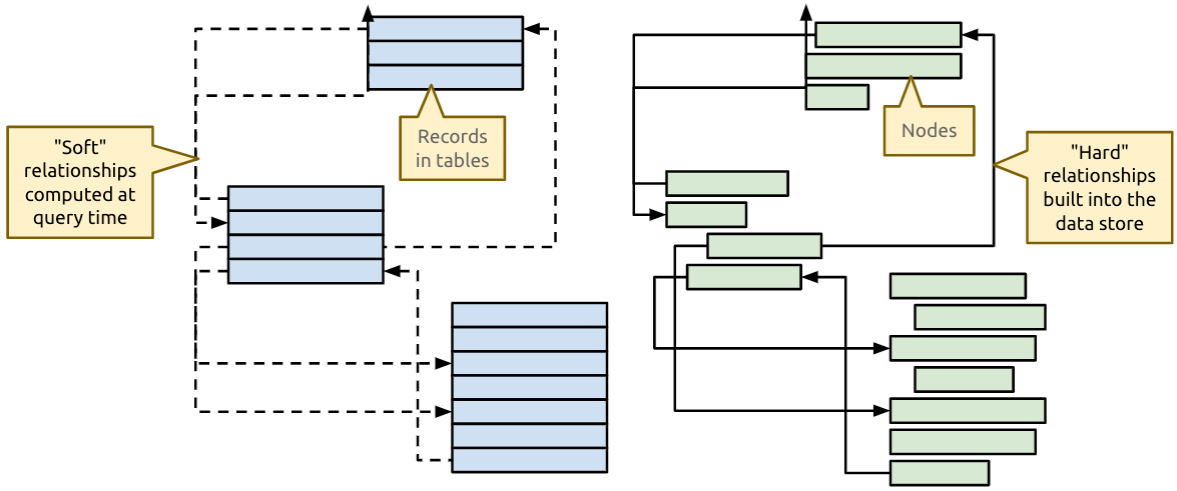
Relational	Graph
Columns	Properties

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.

Run-time behavior: RDBMS vs graph

How data is retrieved is very different between an RDBMS and a graph database:



How we model: RDBMS vs graph

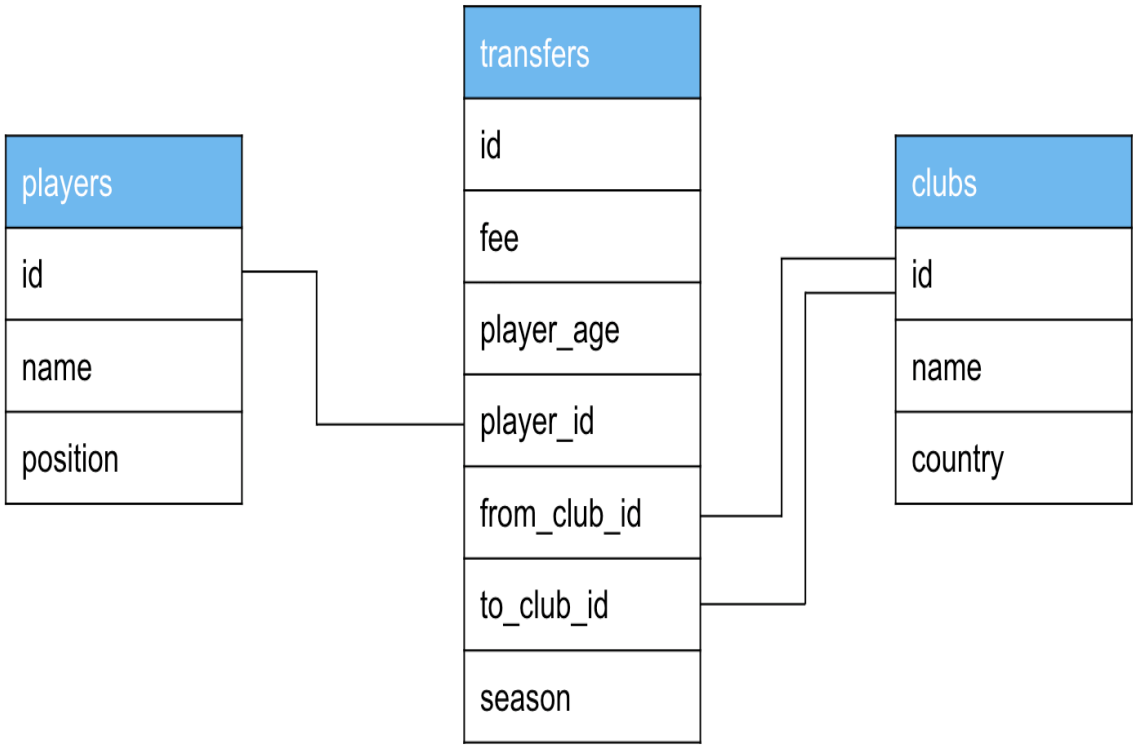
How you model data from relational vs graph differs:

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.

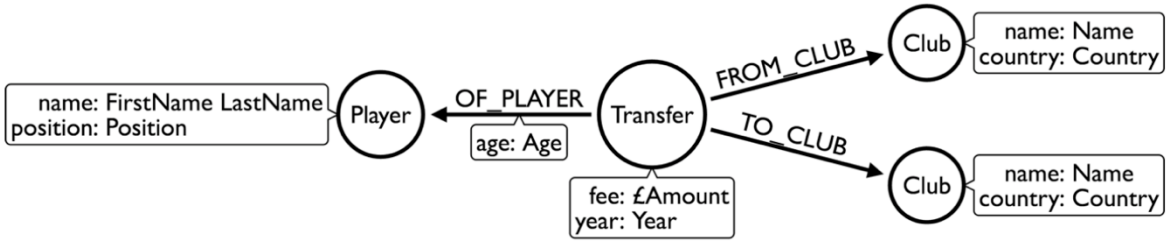
Relational	Graph
More abstract focus when modeling i.e. focus on classes rather than objects.	Common to use actual data items when modeling.

If we were modeling a football transfers graph in relational and graph databases these diagrams show what common approaches might look like.

Here is the relational model:

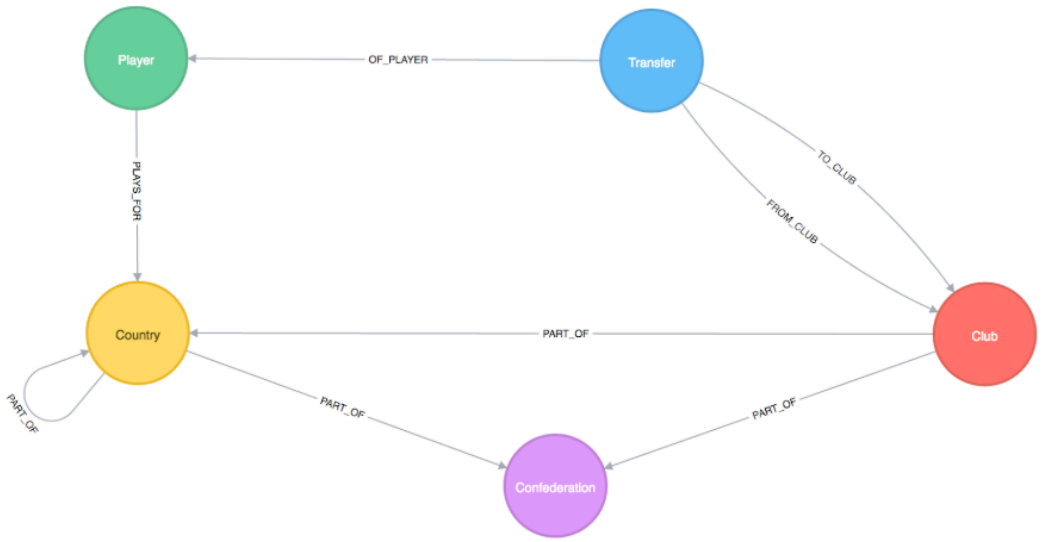


and here is the corresponding graph model:



With the graph model we might sketch out examples with actual values and derive the ‘schema’ while doing that modeling process.

In Neo4j, the data model might evolve to something like this:



How does Neo4j support the property graph model?

- Neo4j is a **Database** – use it to reliably **store information** and **find it later**.
- Neo4j's data model is a **Graph**, in particular a **Property Graph**.
- **Cypher** is Neo4j's graph query language (**SQL for graphs!**).
- Cypher is a declarative query language: it describes **what** you are interested in, not **how** it is acquired.
- Cypher is meant to be very **readable** and **expressive**.

Introduction to Neo4j

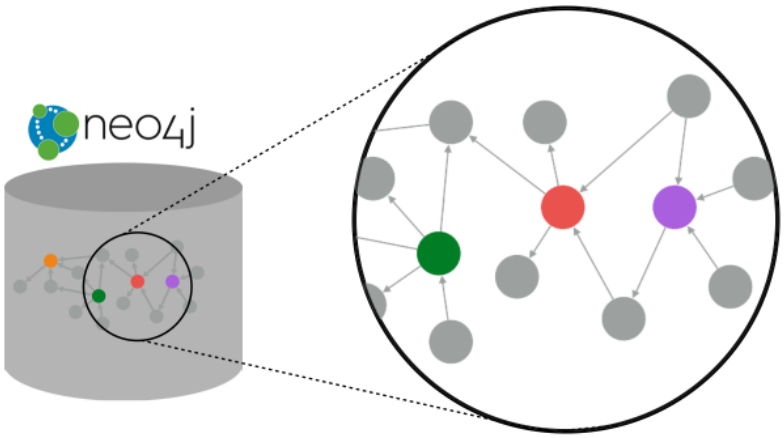
Neo4j Graph Platform

The Neo4j Graph Platform includes components that enable you to develop your graph-enabled application. To better understand the Neo4j Graph Platform, you will learn about these components and the benefits they provide.

Neo4j Database

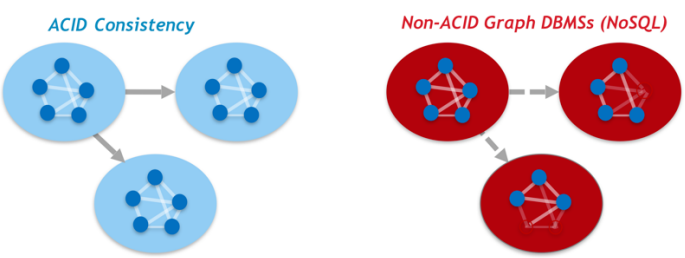
The heart of the Neo4j Graph Platform is the Neo4j Database. The Neo4j Graph Platform includes out-of-the-box tooling that enables you to access graphs in Neo4j Databases. In addition, Neo4j provides APIs and drivers that enable you to create applications and custom tooling for accessing and visualizing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

Neo4j Database: Index-free adjacency



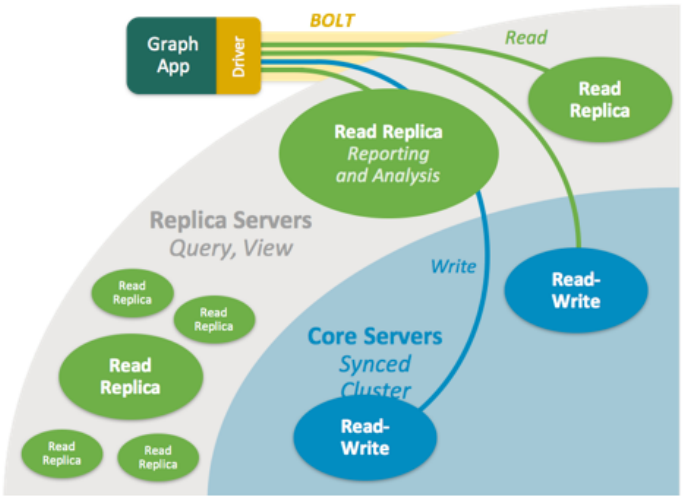
With index free adjacency, when a node or relationship is written to the database, it is stored in the database as connected and any subsequent access to the data is done using pointer navigation which is very fast. Since Neo4j is a native graph database (i.e. it has a graph as its core data model), 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 availablity, 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.

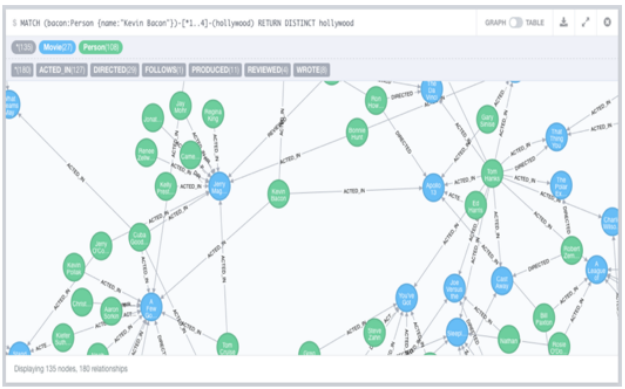
In addition, Neo4j supports Java, JavaScript, Python, C#, and Go drivers out of the box that use Neo4j's bolt protocol for binary access to the database layer. Bolt is an efficient binary protocol that compresses data sent over the wire as well as encrypting the data. For example, you can write a Java 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

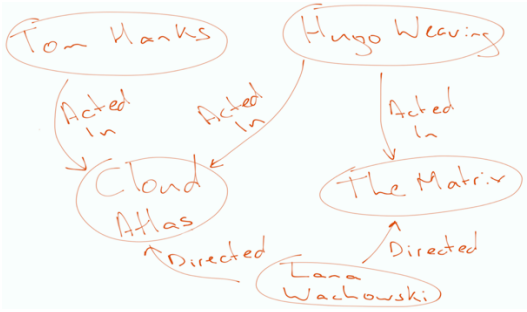


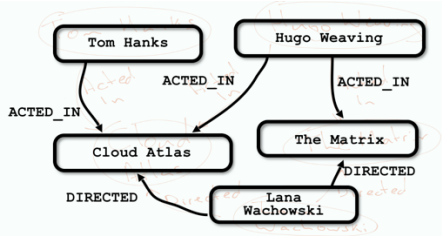
In a development environment, you will use the Neo4j Browser or a Web browser to access data and test your Cypher statements, most of which will be used as part of your application code. Neo4j Browser is an application that uses the JavaScript Bolt driver to access the graph engine of the Neo4j database 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.

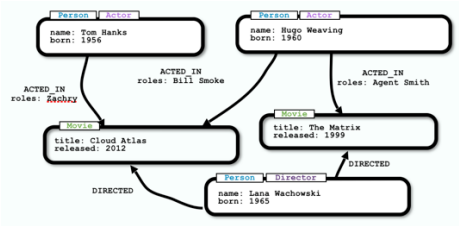
In this video, you can see how Neo4j Bloom can be used to examine and modify a Graph, even when you know very little about Cypher:

Whiteboard modeling

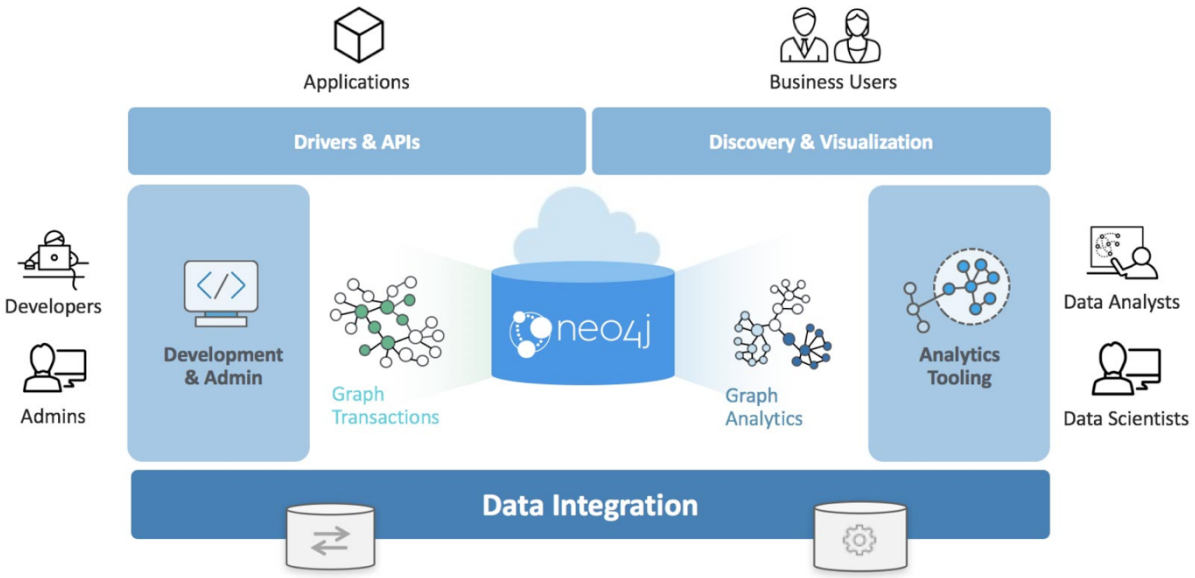
With a property graph model, it is very easy to collaborate with colleagues to come up with a whiteboard model of your data that is easy to understand and easy modify. You then use the model to create the nodes, relationships, labels, and properties you will use for your Neo4j data. Even after the graph has been defined and populated with data, it is easy to modify the graph as your application needs change.


Tom Hanks
Hugo Weaving
Acted In
Acted In
Acted In
Cloud Atlas
The Matrix
Directed
Directed
Lana Wachowski


Tom Hanks
Hugo Weaving
ACTED_IN
ACTED_IN
ACTED_IN
Cloud Atlas
The Matrix
DIRECTED
DIRECTED
Lana Wachowski

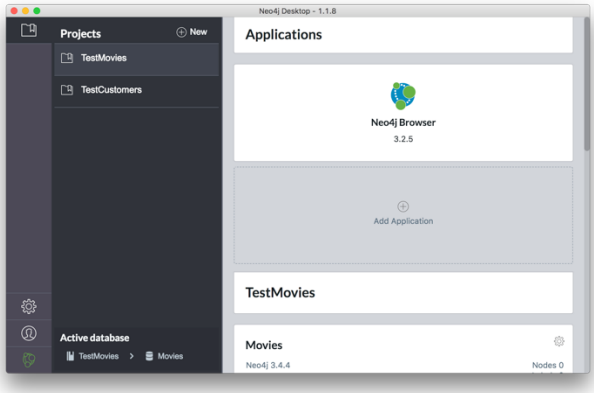

Person
Actor
name: Tom Hanks
born: 1956
Person
Actor
name: Hugo Weaving
born: 1960
ACTED_IN
roles: Bill Smoke
ACTED_IN
roles: Agent Smith
ACTED_IN
Movie
title: Cloud Atlas
released: 2012
Movie
title: The Matrix
released: 1999
DIRECTED
DIRECTED
Person
Director
name: Lana Wachowski
born: 1965

Neo4j Graph Platform architecture



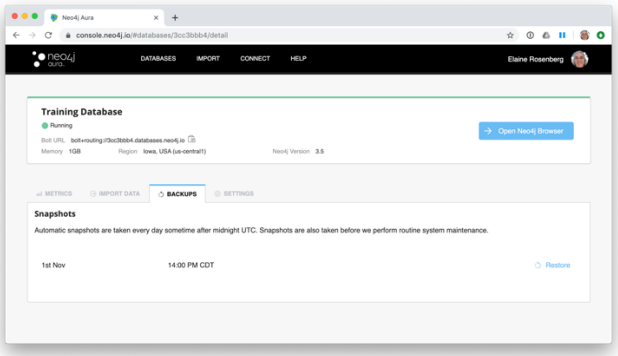
Here is the big picture of the Neo4j Graph Platform. The Neo4j Database provides support for graph transactions and analytics. Developers use the Neo4j Desktop, along with Neo4j Browser to develop graphs and test them, as well as implement their applications in a number of languages using supported drivers, tools and APIs. Administrators use tools to manage and monitor Neo4j Databases and clusters. Business users use out-of-the box graph visualization tools or they use custom tools. Data analysts and scientists use the analytics capabilities in the Graph Algorithm libraries or use custom libraries to understand and report findings to the enterprise. Applications can also integrate with existing databases (SQL or NoSQL), layering Neo4j on top of them to provide rich, graph-enabled access to the data.

Neo4j Desktop



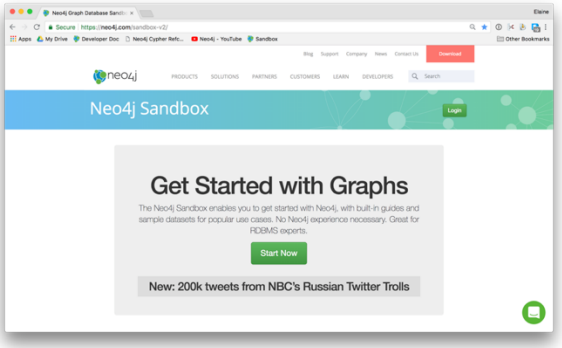
The Neo4j Desktop includes the Neo4j Database server which has a graph engine and kernel so that Cypher statements can be executed to access a database on your system. It includes an application called Neo4j Browser. Neo4j Browser enables you to access a Neo4j database using Cypher. You can also call built-in procedures that communicate with the database server. There are a number of additional libraries and drivers for accessing the Neo4j database from Cypher or from another programming language that you can install in your development environment. If you are looking to use your system for application development and you want to be able to create multiple Neo4j databases on your machine, you should consider downloading the Neo4j Desktop (free download). The Neo4j Desktop runs on OS X, Linux, and Windows.

Neo4j Aura



Neo4j Aura enables you to create a Neo4j Database instance in the Cloud using a monthly subscription model. The amount per month depends on the amount of memory required for the database. This frees you from needing to install Neo4j on your system. Once you create a Neo4j Database at the Neo4j Aura site, it will be managed by Neo4j. Backups are done automatically for you and the database is available 24X7. In addition, the Neo4j will ensure that the database instance is always up-to-date with the latest version of Neo4j.

Neo4j Sandbox



The Neo4j Sandbox is another way that you can begin development with Neo4j. It is a temporary, cloud-based instance of a Neo4j Server with its associated graph that you can access from any Web browser. The database in a Sandbox may be blank or it may be pre-populated. It is started automatically for you when you create the Sandbox.

By default, the Neo4j Sandbox is available for three days, but you can extend it for up to 10 days. If you do not want to install Neo4j Desktop on your system, consider creating a Neo4j Sandbox. You must make sure that you extend your lease of the Sandbox, otherwise you will lose your graph and any saved Cypher scripts you have created in the Sandbox. However, you can use Neo4j Browser Sync to save Cypher scripts from your Sandbox. We recommend you use the Neo4j Desktop or Neo4j Aura for a real development project. The Sandbox is intended as a temporary environment or for learning about the features of Neo4j as well as specific graph use-cases.

Steps for setting up your development environment for this training

If you are using Neo4j Desktop:

1. Install Neo4j Desktop.
2. In a project, create a local graph (database).
3. Start the database.
4. Click the Neo4j Browser application.

Introduction to Cypher

Cypher is the query language you use to retrieve data from the Neo4j Database, as well as create and update the data. Check the manual and reference at:

- Neo4j Cypher Manual
- Cypher Reference card

What is Cypher?

Cypher is a declarative query language that allows for expressive and efficient querying and updating of graph data. Cypher is a relatively simple and very powerful language. Complex database queries can easily be expressed through Cypher, allowing you to focus on your domain instead of getting lost in the syntax of database access.

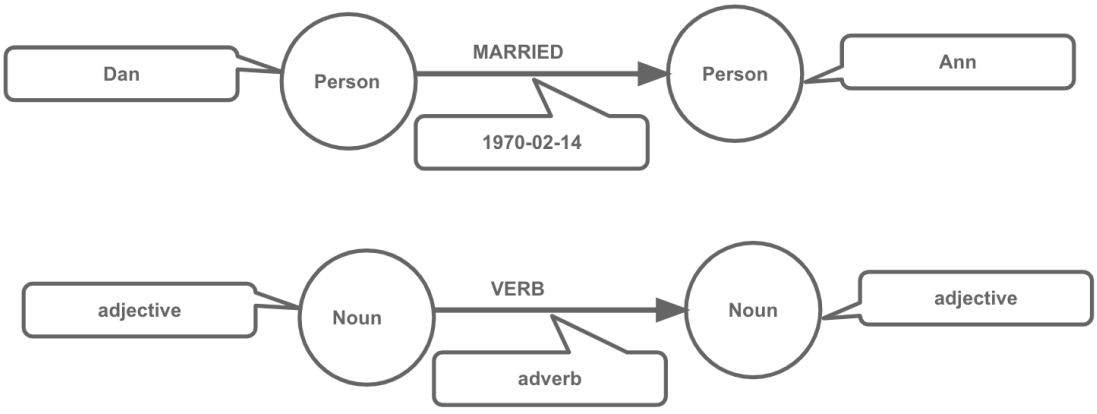
Cypher is designed to be a human-friendly query language, suitable for both developers and other professionals. The guiding goal is to make the simple things easy, and the complex things possible.

Cypher is ASCII art

Optimized for being read by humans, Cypher's construct uses English prose and iconography (called ASCII Art) to make queries more self-explanatory.

```
(A)-[:LIKES]->(B),(A)-[:LIKES]->(C),(B)-[:LIKES]->(C)

    (A)-[:LIKES]->(B)-[:LIKES]->(C)<-[:LIKES]-(A)
```

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.



This is in contrast to imperative, programmatic APIs for database access. This approach makes query optimization an implementation detail instead of a burden on the developer, removing the requirement to update all traversals just because the physical database structure has changed.

Cypher is inspired by a number of different approaches and builds upon established practices for expressive querying. Many of the Cypher keywords like `WHERE` and `ORDER BY` are inspired by SQL. The pattern matching functionality of Cypher borrows concepts from SPARQL. And some of the collection semantics have been borrowed from languages such as Haskell and Python.

The Cypher language has been made available to anyone to implement and use via openCypher (opencypher.org), allowing any database vendor, researcher or other interested party to reap the benefits of our years of effort and experience in developing a first class graph query language.

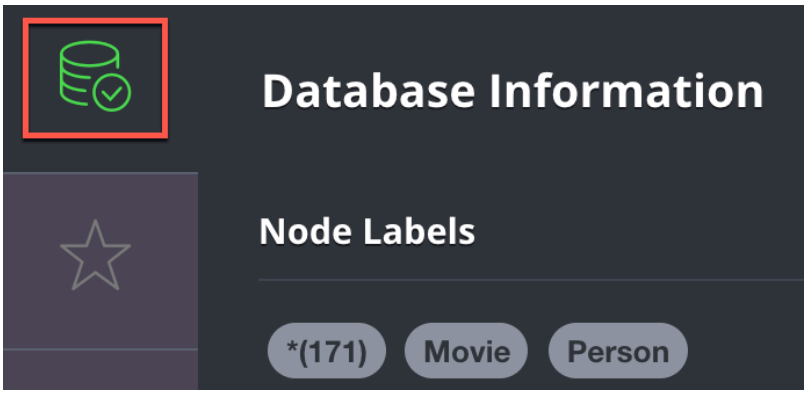
Nodes

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.

For example, you can see the labels in the database by simply clicking the Database icon in Neo4j Browser:



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*.

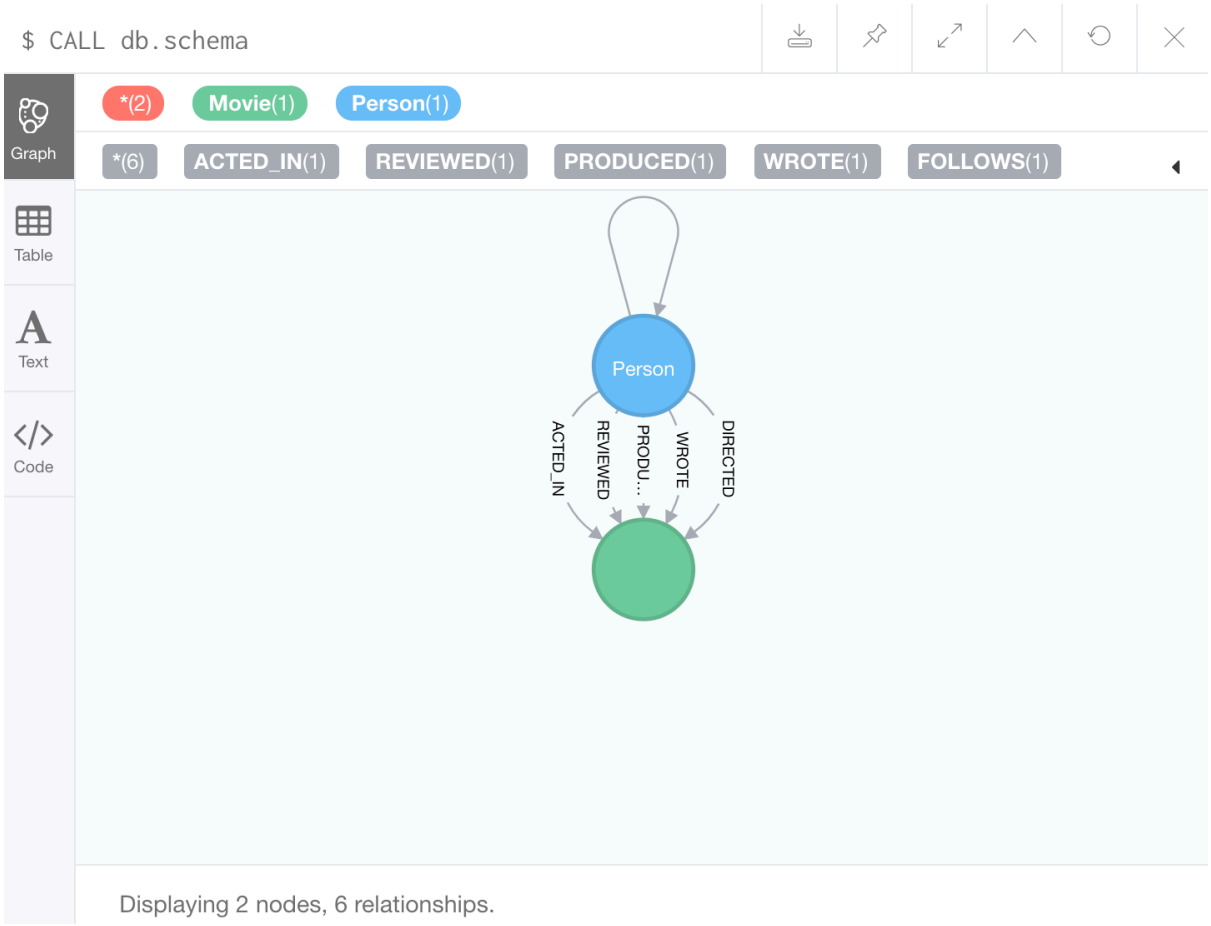
Comments in Cypher

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.

For example, when we run this procedure in our training environment, we see the following in the result pane. Here we see that the graph has 2 labels defined for nodes, *Person* and *Movie*. Each type of nodes is displayed in a different color. The relationships between nodes are also displayed, which you will learn about later in this module.



Using MATCH to retrieve nodes

In this video, you will be introduced to using the `MATCH` statement to retrieve nodes from the graph in Neo4j Browser.

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. Later in this training, you will learn how to use `MATCH` to select nodes and data for updating the graph. First, you will learn how to simply return nodes.

Here are simplified syntax examples for a query:

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

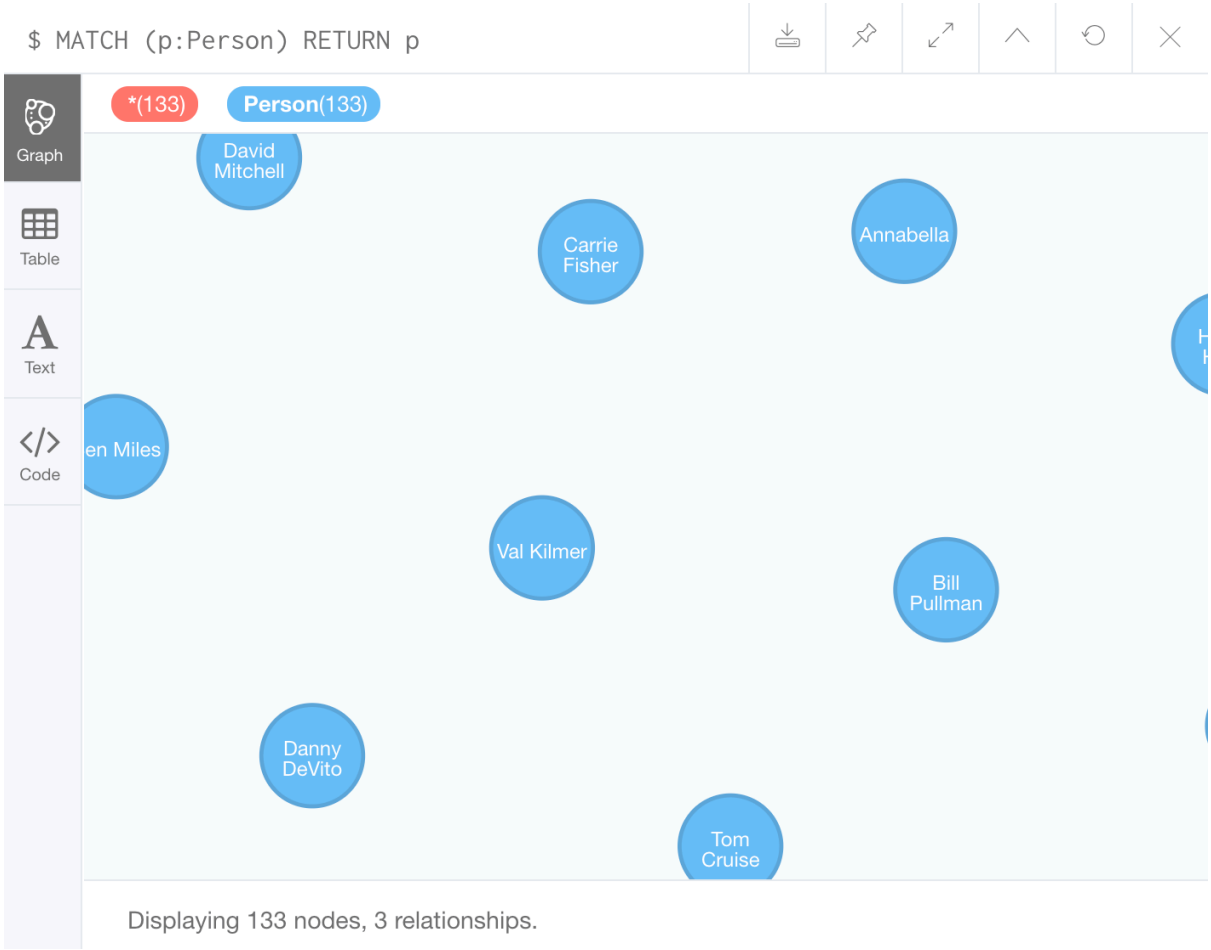
Notice that the Cypher keywords `MATCH` and `RETURN` are upper-case. This coding convention is described in the *Cypher Style Guide* and will be used in this training. This `MATCH` clause returns all nodes in the graph, where the optional *Label* is used to return a subgraph if the graph contains nodes 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:

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

When we execute the Cypher statement, `MATCH (p:Person) RETURN p`, the graph engine returns all nodes with the label *Person*. The default view of the returned nodes are the nodes that were referenced by the variable *p*.

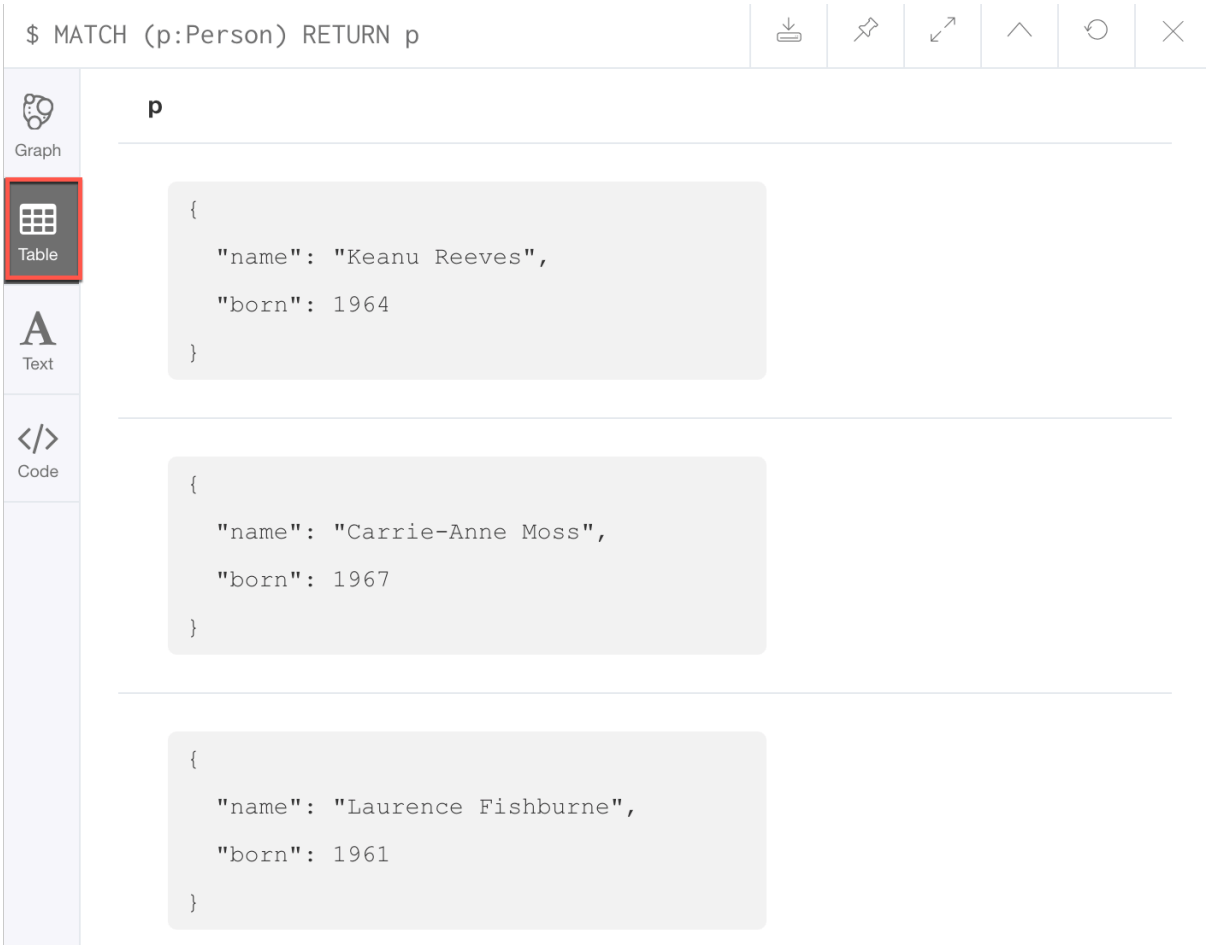
The result returned is:



NOTE	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`.

Viewing nodes as table data

We can also view the nodes as table data where the nodes and their associated property values are shown in a JSON-style format:



When nodes are displayed as table values, the node labels and ids are not shown, only the property values for the nodes. Node ids are unique identifiers and are set by the graph engine when a node is created.

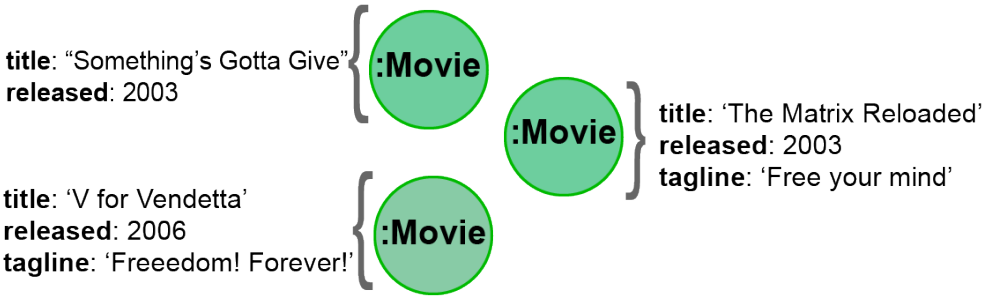
Exercise 1: Retrieving nodes

In the query edit pane of Neo4j Browser, execute the browser command: `:play intro-neo4j-exercises` and follow the instructions for Exercise 1.

Properties

In Neo4j, a node (and a relationship, which you will learn about later) can have properties that are used for 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.

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



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.

Here is what you will see in the result pane when you call the method to return the property keys in the *Movie* graph. This result stream contains all property keys in the graph. It does not display which nodes utilize these property keys.

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.

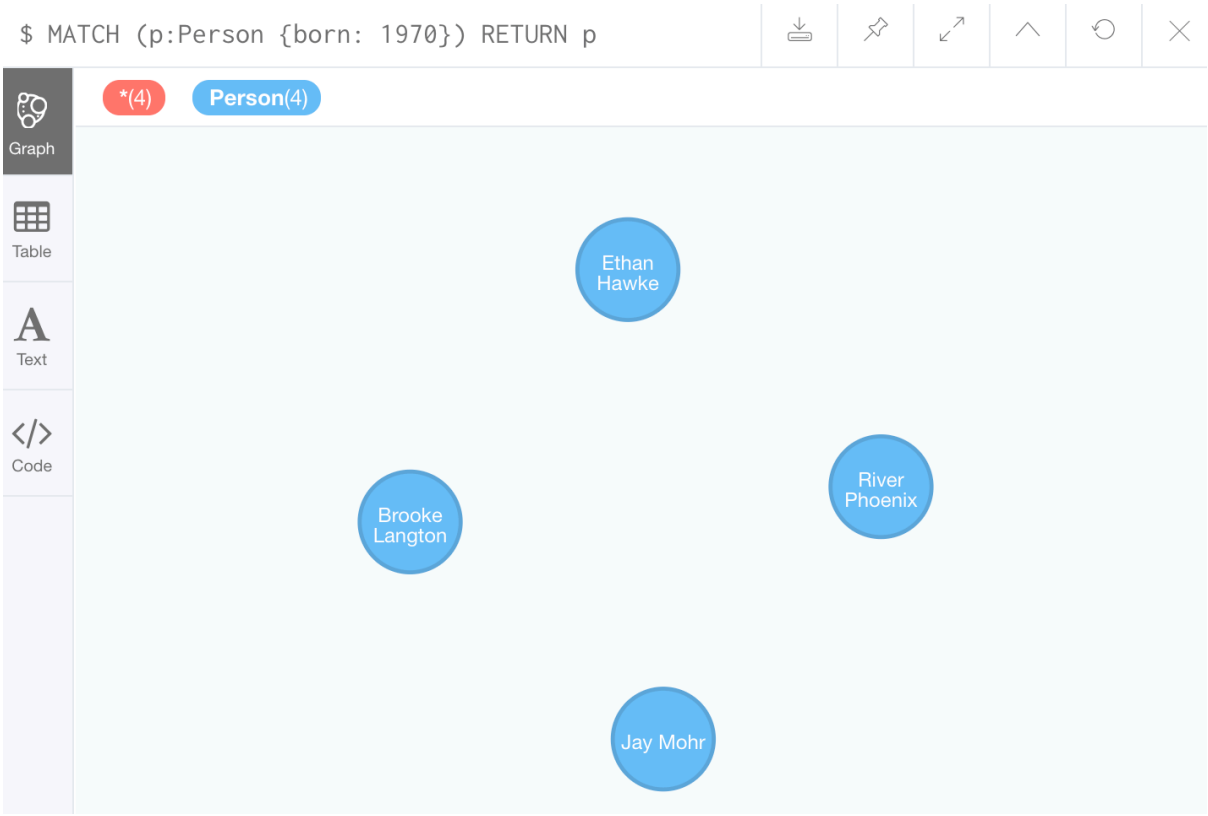
Here are simplified syntax examples for a query where we specify one or more values for properties that will be used to filter the query results and return a subset of the graph:

```
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
```

The result returned is:



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

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

Here is the result returned:

As it turns out, there is only one movie with the *tagline*, *Free your mind* in the *Movie* database, but if we had specified a different year, the query would not have returned a value because when you specify properties, both properties must match.

Returning property values

In this video, you will see how to return property values to the output stream when you retrieve nodes from the graph in Neo4j Browser.

Thus far, you have seen how to retrieve nodes and return nodes (entire graph or a subset of the graph). You can use the `RETURN` clause to return property values of nodes retrieved.

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
```

The result returned is:

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
```

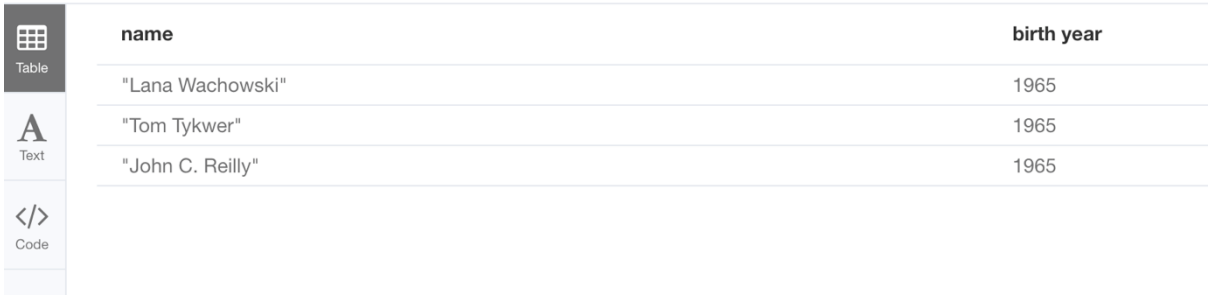
NOTE	If you want a heading to contain a space between strings, you must specify the alias with the back tick `character, rather than a single or double quote character. In fact, you can specify any variable, label, relationship type, or property key with a space also by using the back tick `character.

Here we specify aliases for the returned property values:

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

The result returned is:

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



Exercise 2: Filtering queries using property values

In the query edit pane of Neo4j Browser, execute the browser command: `:play intro-neo4j-exercises` and follow the instructions for Exercise 2.

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.

ASCII art

Thus far, you have learned how to specify a node in a `MATCH` clause. You can specify nodes and their relationships to traverse the graph and quickly find the data of interest.

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
```

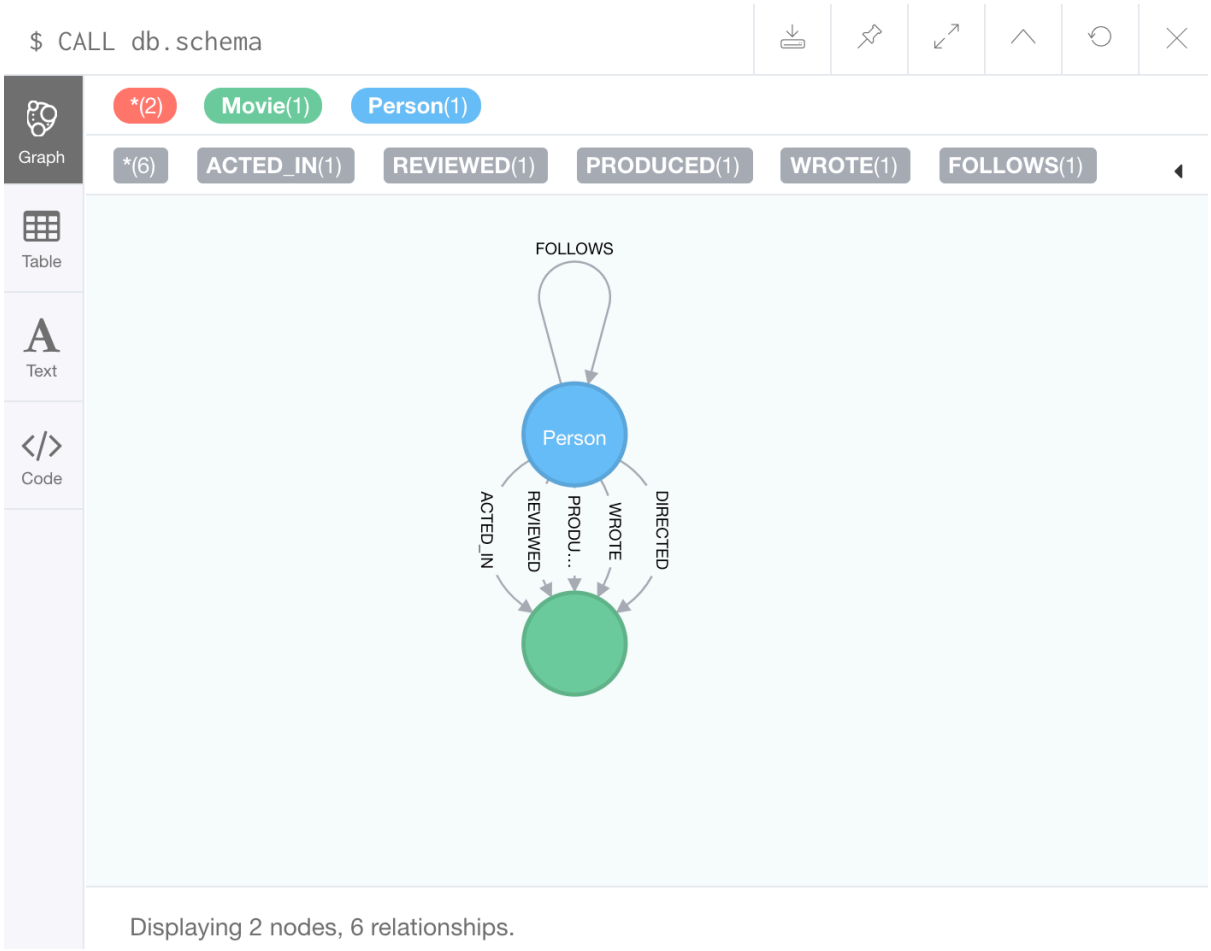
where:

node1	is a specification of a node where you may include node labels and property values for filtering.
:REL_TYPE	is the type (name) for the relationship. For this syntax the relationship is from *node1* to *node2*.
:REL_TYPEA , *:REL_TYPEB*	are the relationships from *node1* to *node2.* The nodes are returned if at least one of the relationships exists.
node2	is a specification of a node where you may include node labels and property values for filtering.

Examining relationships

You can run `CALL db.schema` to view the relationship types in the graph. In the *Movie* graph, we see these relationships between the nodes:

Here we see that this graph has a total of 6 relationship types between the nodes. Some *Person* nodes are connected to other *Person* nodes using the *FOLLOWS* relationship type. All of the other relationships in this graph are from *Person* nodes to *Movie* nodes.



The relationship types can also be viewed by selecting the arrow to the right in the relationship type row.

Querying with properties

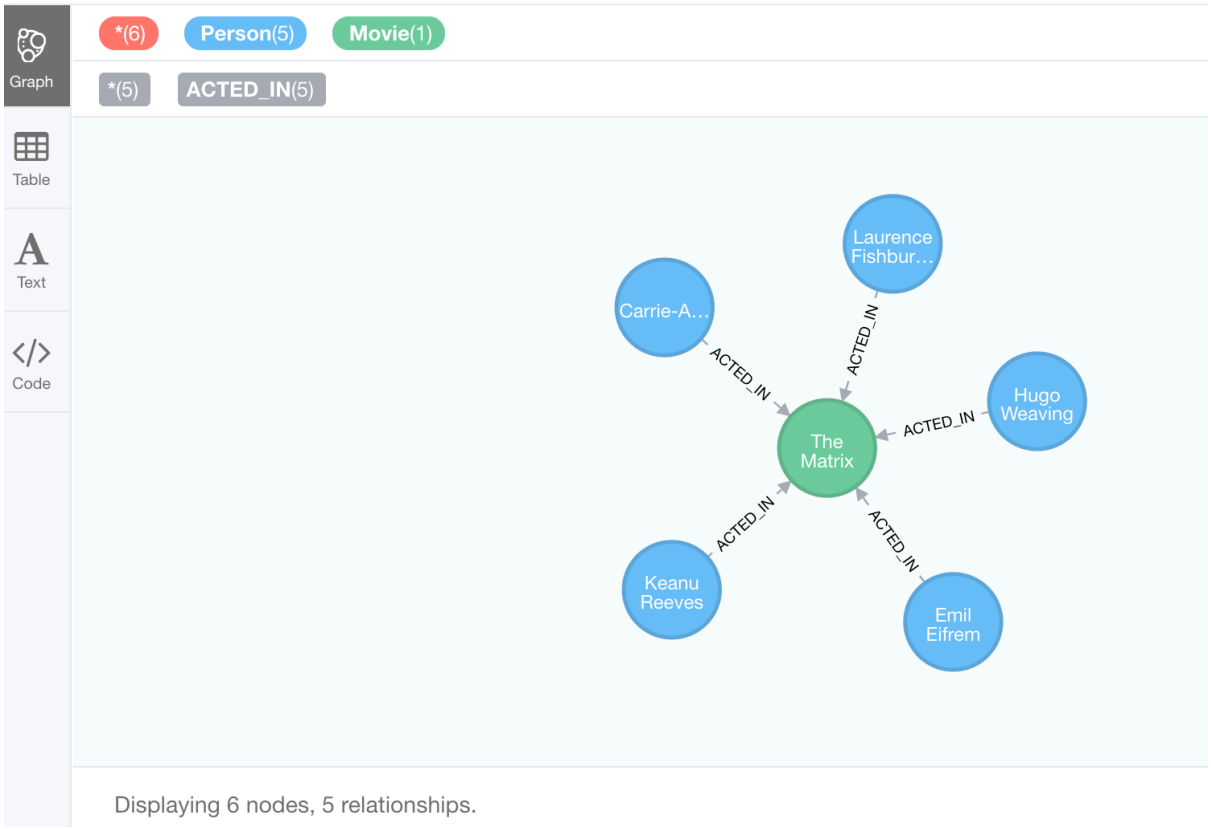
Using a relationship in a query

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
```

The result returned is:

```
$ 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
```

The result returned is:

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

Table | Text | Code

p.name	m.title
"Tom Hanks"	"Apollo 13"
"Tom Hanks"	"Cast Away"
"Tom Hanks"	"The Polar Express"
"Tom Hanks"	"A League of Their Own"
"Tom Hanks"	"Charlie Wilson's War"
"Tom Hanks"	"Cloud Atlas"
"Tom Hanks"	"The Da Vinci Code"
"Tom Hanks"	"The Green Mile"
"Tom Hanks"	"You've Got Mail"
"Tom Hanks"	"That Thing You Do"
"Tom Hanks"	"That Thing You Do"
"Tom Hanks"	"Joe Versus the Volcano"
"Tom Hanks"	"Sleepless in Seattle"

Started streaming 13 records after 1 ms and completed after 1 ms.

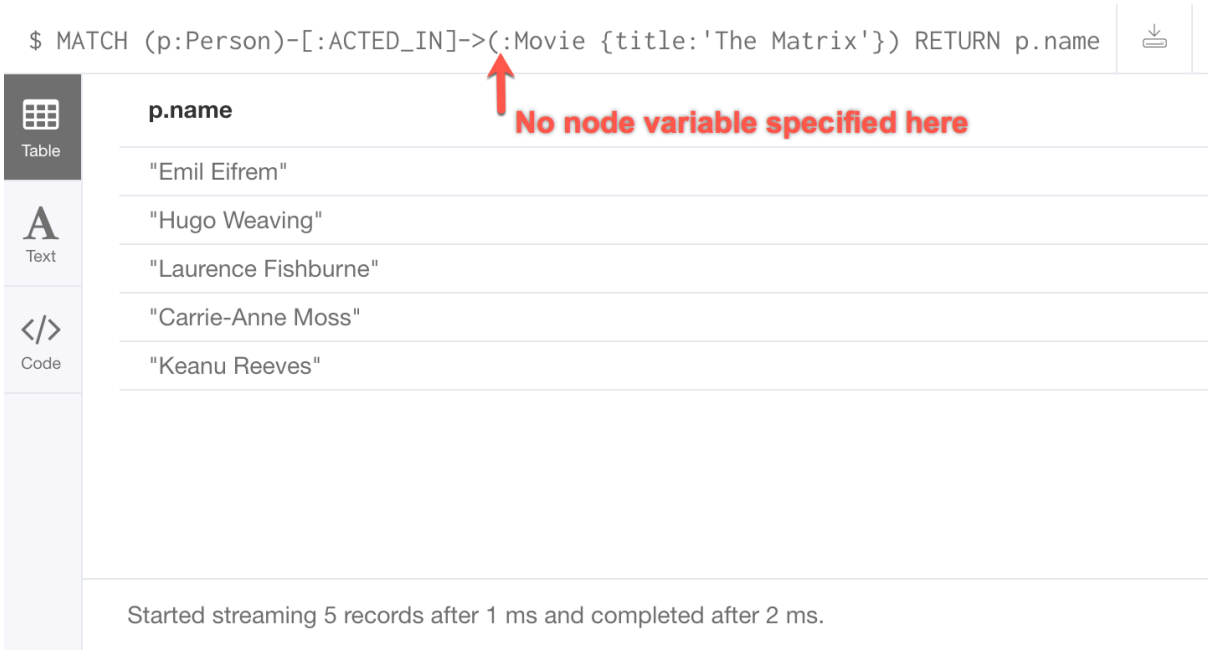
Notice that there are multiple rows returned for the movie, *That Thing You Do*. This is because *Tom Hanks* acted in and directed that movie.

Using anonymous nodes in a query

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
```

The result returned is:



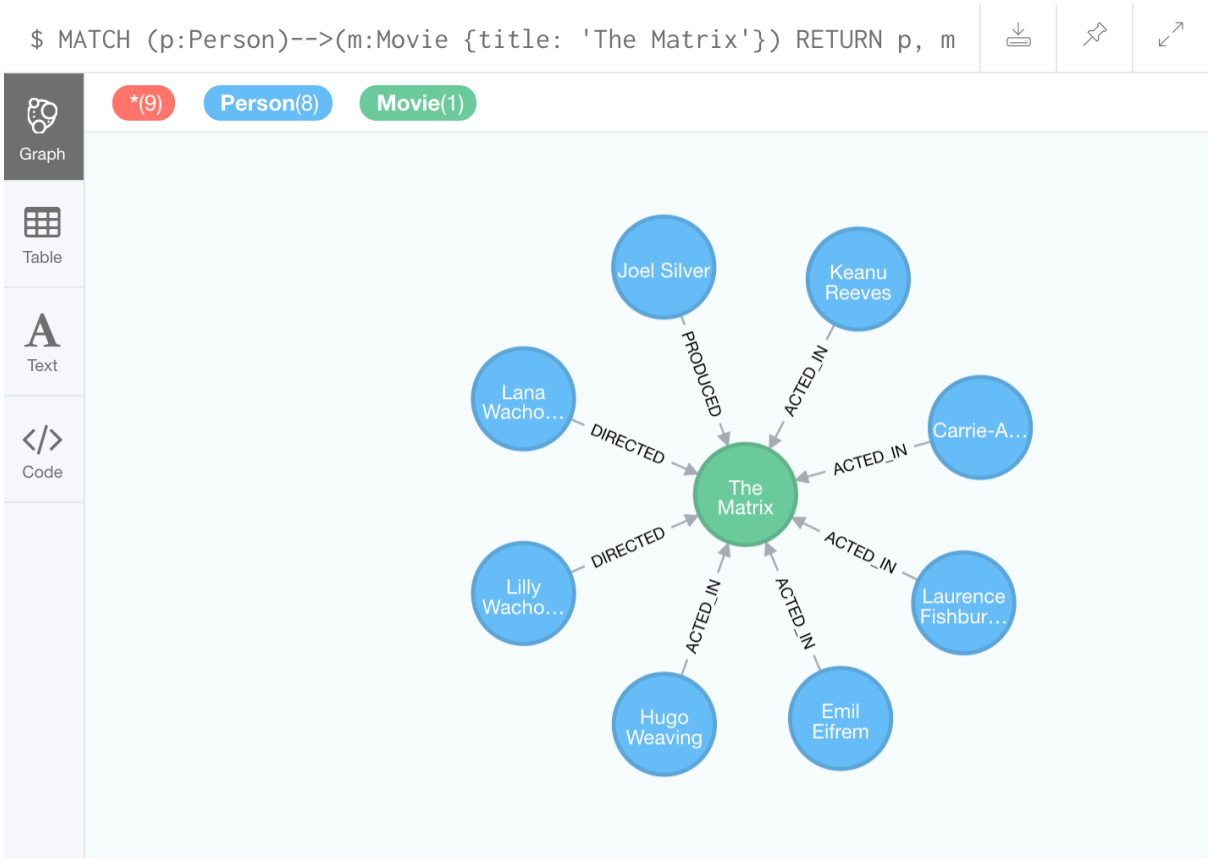
NOTE	A best practice is to place named nodes (those with variables) before anonymous nodes in a `MATCH` clause.

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
```

The result returned is:



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
```

In this training, we will use `-->`, `--`, and `<--` to represent anonymous relationships as it is a Cypher best practice.

Retrieving the relationship types

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

Here is an example where we use the *rel* variable to hold the relationships retrieved. We then use this variable to return the relationship types.

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

The result returned is:

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

Table | Text | Code

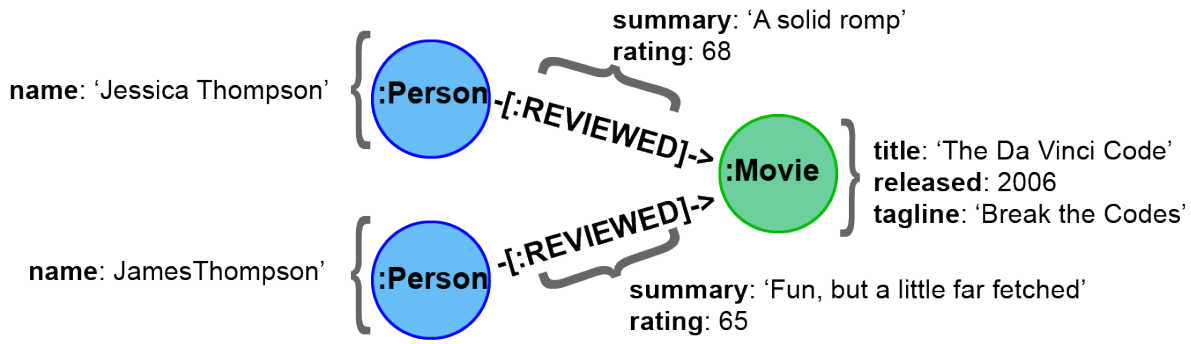
p.name	type(rel)
"Emil Eifrem"	"ACTED_IN"
"Joel Silver"	"PRODUCED"
"Lana Wachowski"	"DIRECTED"
"Lilly Wachowski"	"DIRECTED"
"Hugo Weaving"	"ACTED_IN"
"Laurence Fishburne"	"ACTED_IN"
"Carrie-Anne Moss"	"ACTED_IN"
"Keanu Reeves"	"ACTED_IN"

Started streaming 8 records after 1 ms and completed after 2 ms.

Retrieving properties for relationships

Recall that a node can have as 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.

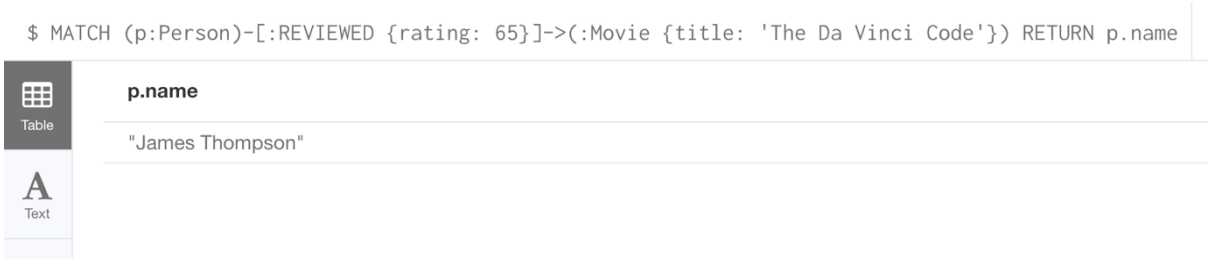
Here is an example from the *Movie* graph. The movie, *The Da Vinci Code* has two people that reviewed it, *Jessica Thompson* and *James Thompson*. Each of these *Person* nodes has the *REVIEWED* relationship to the *Movie* node for *The Da Vinci Code*. Each relationship has properties that further describe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



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 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
```

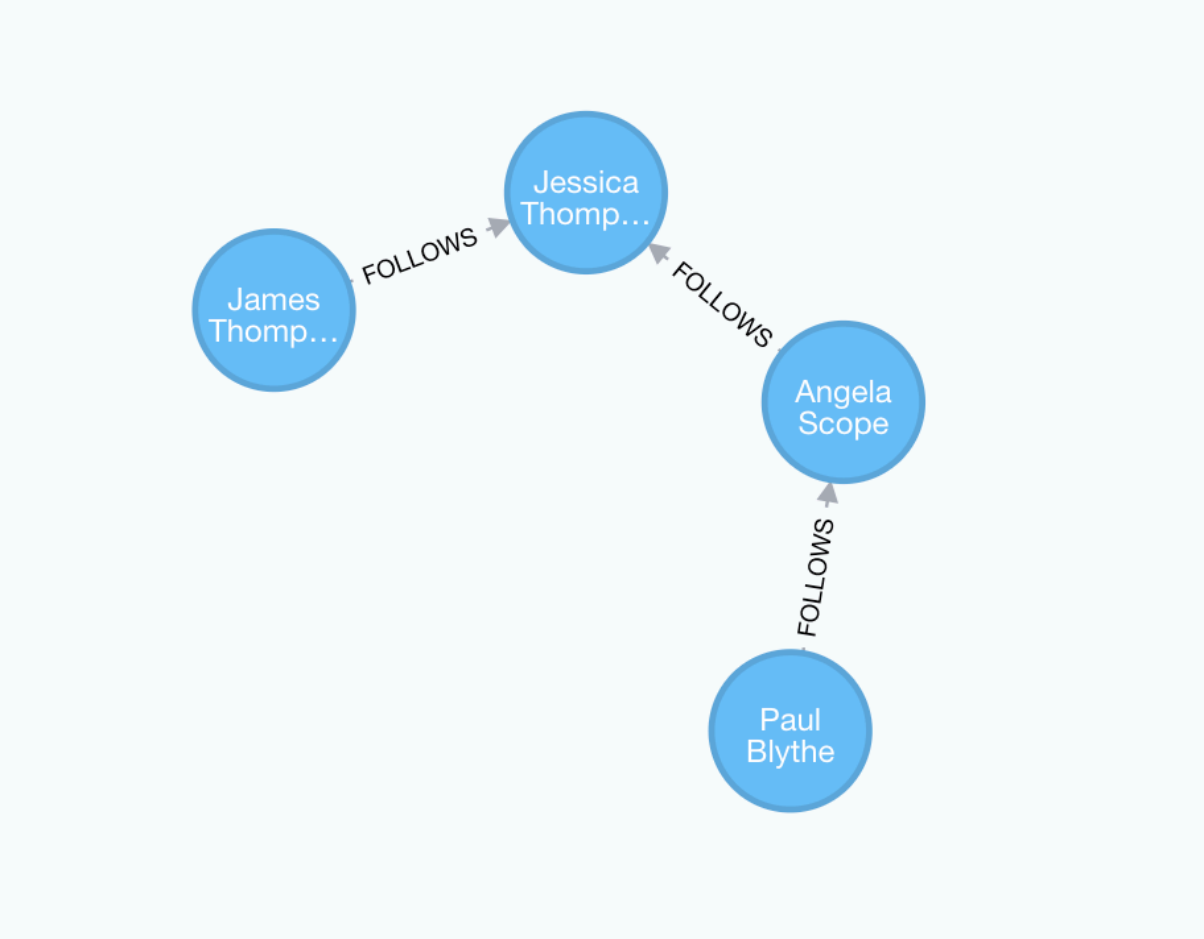
The result returned is:



Using patterns for queries

Thus far, you have learned how to specify nodes, properties, and relationships in your Cypher 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.

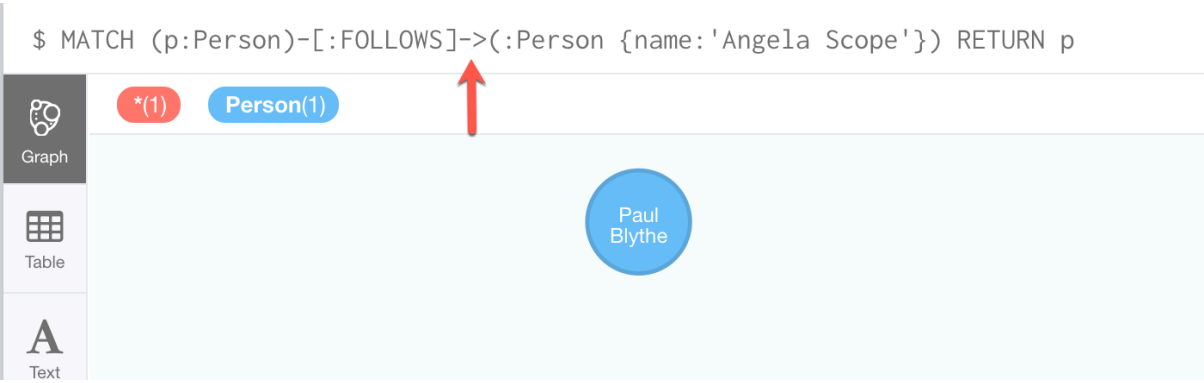
Here is an example of where the *FOLLOWS* relationship is used in the *Movie* graph. Notice that this relationship is directional.



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

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

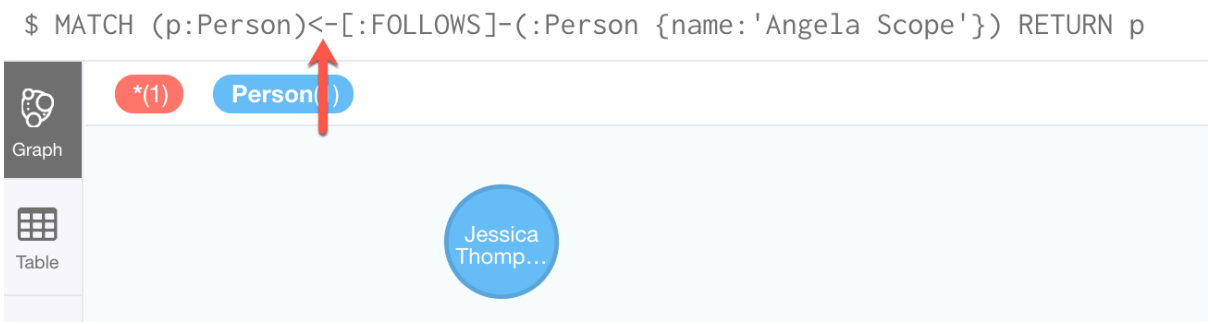
The result returned is:



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

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

The result returned is:

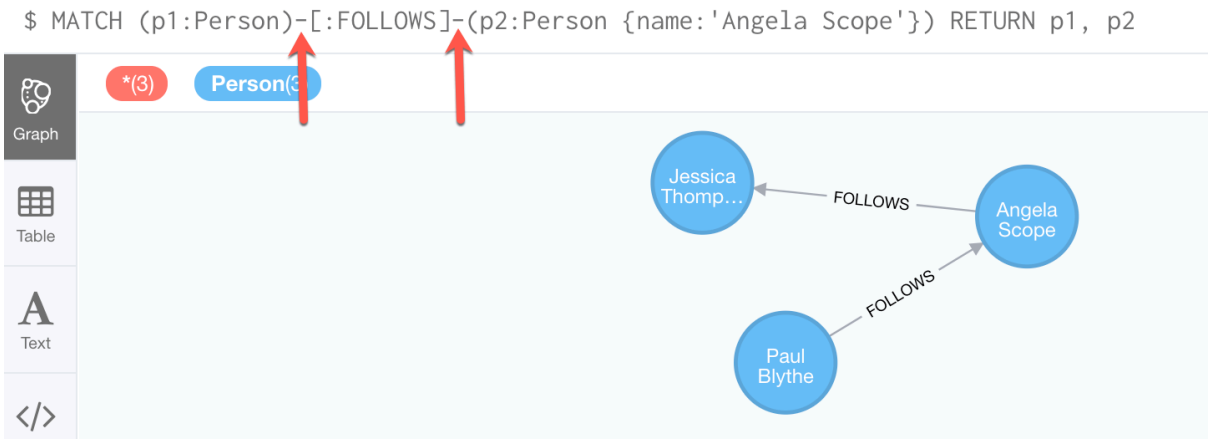


Querying by any direction of the relationship

We can also find out what *Person* nodes are connected by the *FOLLOWED* relationship in either direction by removing the directional arrow from the pattern.

```
MATCH  (p1:Person)-[:FOLLOWS]-(p2:Person {name:'Angela Scope'})
RETURN p1, p2
```

The result returned is:



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*.

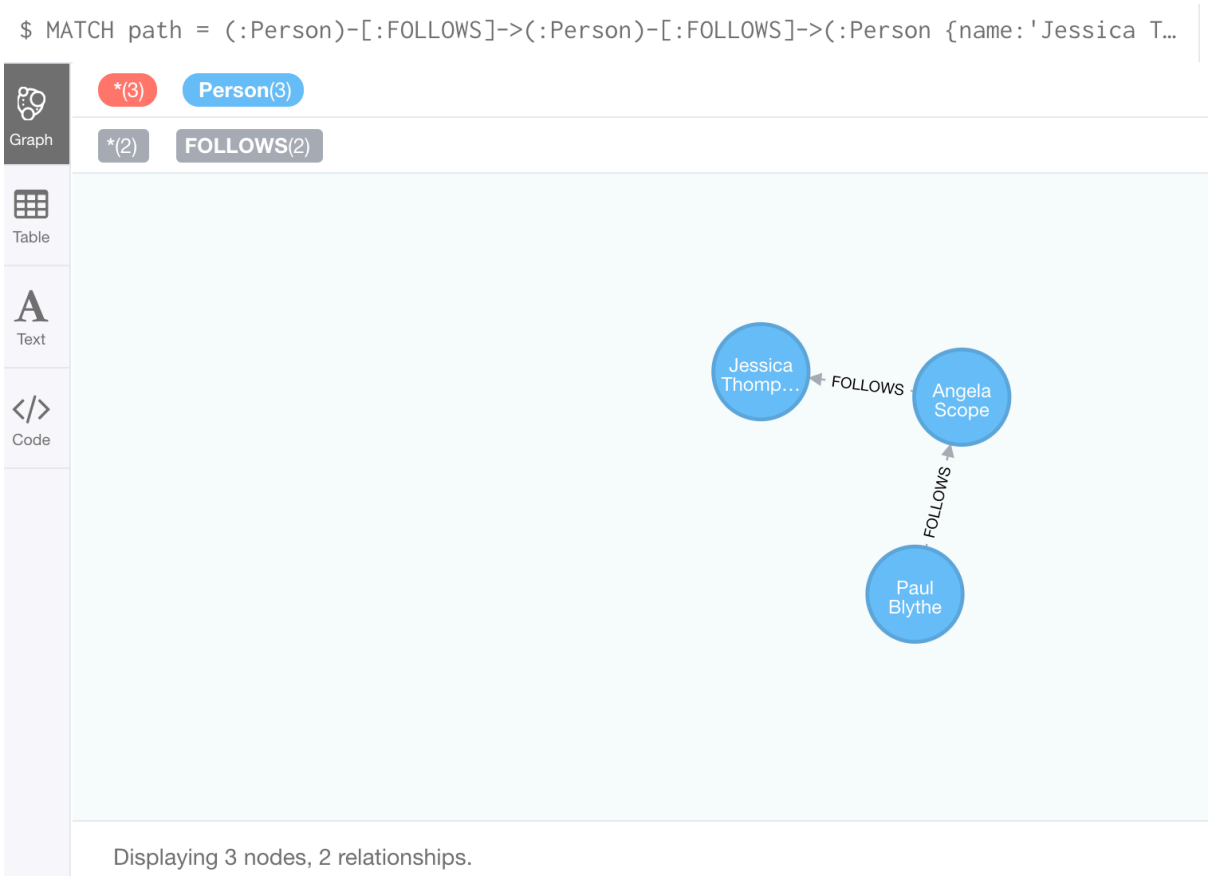
```
MATCH  (p:Person)-[:FOLLOWS]->(:Person)-[:FOLLOWS]->(:Person {name:'Jessica Thompson'})
RETURN p
```

The result returned is:

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
```

The result returned is:



Using relationship direction to optimize a query

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
```

The result returned is:

a.name	m.title	d.name
"Emil Eifrem"	"The Matrix"	"Lana Wachowski"
"Hugo Weaving"	"The Matrix"	"Lana Wachowski"
"Laurence Fishburne"	"The Matrix"	"Lana Wachowski"
"Carrie-Anne Moss"	"The Matrix"	"Lana Wachowski"
"Keanu Reeves"	"The Matrix"	"Lana Wachowski"
"Emil Eifrem"	"The Matrix"	"Lilly Wachowski"
"Hugo Weaving"	"The Matrix"	"Lilly Wachowski"
"Laurence Fishburne"	"The Matrix"	"Lilly Wachowski"
"Carrie-Anne Moss"	"The Matrix"	"Lilly Wachowski"
"Keanu Reeves"	"The Matrix"	"Lilly Wachowski"
"Hugo Weaving"	"The Matrix Reloaded"	"Lana Wachowski"
"Laurence Fishburne"	"The Matrix Reloaded"	"Lana Wachowski"
"Carrie-Anne Moss"	"The Matrix Reloaded"	"Lana Wachowski"
"Keanu Reeves"	"The Matrix Reloaded"	"Lana Wachowski"

In this query, notice that there are multiple records returned for a movie, each with its set of values for the actor and director.

Later in this training, you will learn other ways to query data and how to control the results returned.

Cypher style recommendations

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 `<--`

Here is an example showing some best coding practices:

```
MATCH (:Person {name: 'Diane Keaton'})-[movRel:ACTED_IN]->
(:Movie {title:"Something's Gotta Give"})
RETURN movRel.roles
```

We recommend that you follow the Cypher Style Guide when writing your Cypher statements.

Exercise 3: Filtering queries using relationships

In the query edit pane of Neo4j Browser, execute the browser command: `:play intro-neo4j-exercises` and follow the instructions for Exercise 3.

Getting More Out of Queries

Filtering queries using `WHERE`

You have learned how to specify values for properties of nodes and relationships to filter what data is returned from the `MATCH` and `RETURN` clauses. 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 you 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.

Previously, you learned to write simple query as follows:

```
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
```

In this example, you can only refer to named nodes or relationships in a `WHERE` clause so remember that you must specify a variable for any node or relationship you are testing in the `WHERE` clause. The benefit of using a `WHERE` clause is that you can specify potentially complex conditions for the query.

For example:

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

Specifying ranges in WHERE clauses

Not only can the equality `=` be tested, but you can test ranges, existence, strings, as well as specify logical operations during the query.

Here is an example of specifying a range for filtering the query:

```
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 result:

p.name	m.title	m.released
"Carrie-Anne Moss"	"The Matrix Reloaded"	2003
"Laurence Fishburne"	"The Matrix Reloaded"	2003
"Keanu Reeves"	"The Matrix Reloaded"	2003
"Hugo Weaving"	"The Matrix Reloaded"	2003
"Laurence Fishburne"	"The Matrix Revolutions"	2003
"Hugo Weaving"	"The Matrix Revolutions"	2003
"Keanu Reeves"	"The Matrix Revolutions"	2003
"Carrie-Anne Moss"	"The Matrix Revolutions"	2003
"Jack Nicholson"	"Something's Gotta Give"	2003
"Diane Keaton"	"Something's Gotta Give"	2003
"Keanu Reeves"	"Something's Gotta Give"	2003
"Tom Hanks"	"The Polar Express"	2004

Started streaming 12 records after 1 ms and completed after 8 ms.

You can also specify the same query as:

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

You can specify conditions in a `WHERE` clause that return a value of `true` or `false` (for example predicates). For testing numeric values, you use the standard numeric comparison operators. Each condition can be combined for runtime evaluation using the boolean operators `AND`, `OR`, `XOR`, and `NOT`. There are a number of numeric functions you can use in your conditions. See the *Neo4j Cypher Manual's* section *Mathematical Functions* for more information.

A special condition in a query is when the retrieval returns an unknown value called `null`. You should read the *Neo4j Cypher Manual's* section *Working with null* to understand how `null` values are used at runtime.

Testing labels

Thus far, you have used the node labels to filter queries in a `MATCH` clause. You can filter node labels in the `WHERE` clause also:

For example, these two Cypher queries:

```
MATCH (p:Person)
RETURN p.name
```

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

can be rewritten using `WHERE` clauses as follows:

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

Not all node labels need to be tested during a query, but if your graph has multiple labels for the same node, filtering it by the node label will provide better query performance.

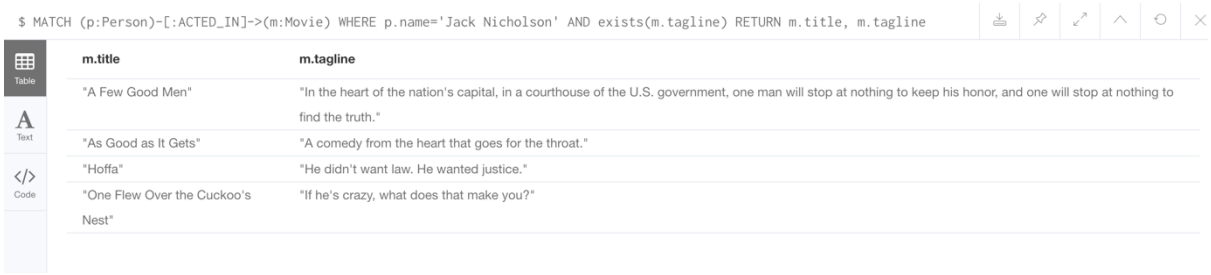
Testing the existence of a property

Recall that a property is associated with a particular node or relationship. A property is not associated with a node with a particular label or relationship type. In one of our queries earlier, we saw that the movie “Something’s Gotta Give” is the only movie in the *Movie* database that does not have a *tagline* 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.

Here is the query to retrieve the specified movies where we test the existence of the *tagline* property:

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

Here is the result:



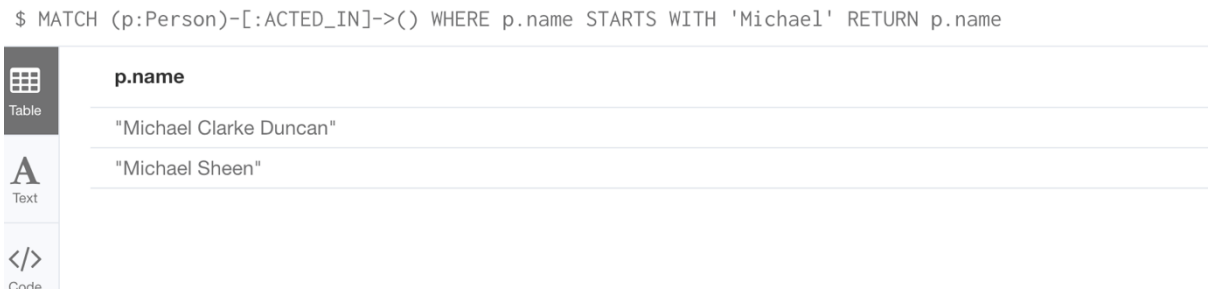
Testing strings

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`.

For example, to find all actors in the *Movie* database whose first name is *Michael*, you would write:

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

Here is the result:



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
```

NOTE	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.

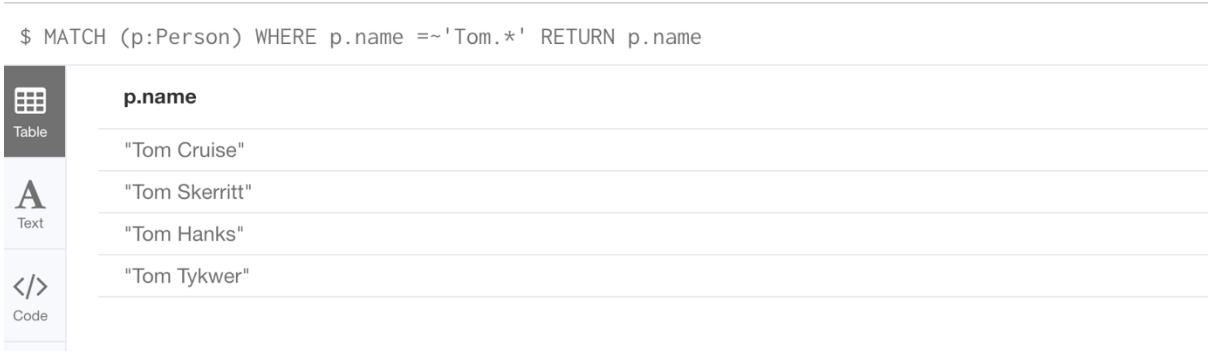
See the *String functions* section of the *Neo4j Cypher Manual* for more information. It is sometimes useful to use the built-in string functions to modify the data that is returned in the query in the RETURN clause.

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
```

Here is the result:



NOTE	If you specify a regular expression. The index will never be used. In addition, the property value must fully match the regular expression.

Testing with patterns

Sometimes during a query, you may want to perform additional filtering using the relationships between nodes being visited during the query. For example, during retrieval, you may want to exclude certain paths traversed. You can specify a `NOT` specifier on a pattern in a `WHERE` clause.

Here is an example where we want to return all *Person* nodes of people who wrote movies:

```
MATCH (p:Person)-[:WROTE]->(m:Movie)
RETURN p.name, m.title
```

Here is the result:

```
$ MATCH (p:Person)-[:WROTE]->(m:Movie) RETURN p.name, m.title
```

Table | Text | Code

p.name	m.title
"Aaron Sorkin"	"A Few Good Men"
"Jim Cash"	"Top Gun"
"Cameron Crowe"	"Jerry Maguire"
"Nora Ephron"	"When Harry Met Sally"
"David Mitchell"	"Cloud Atlas"
"Lilly Wachowski"	"V for Vendetta"
"Lana Wachowski"	"V for Vendetta"
"Lana Wachowski"	"Speed Racer"
"Lilly Wachowski"	"Speed Racer"
"Nancy Meyers"	"Something's Gotta Give"

Started streaming 10 records in less than 1 ms and completed after 1 ms.

Next, we modify this query to exclude people who directed that movie:

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

Here is the result:

```
$ MATCH (p:Person)-[:WROTE]->(m:Movie) WHERE NOT exists( (p)-[:DIRECTED]->(m) ) RETURN …
```

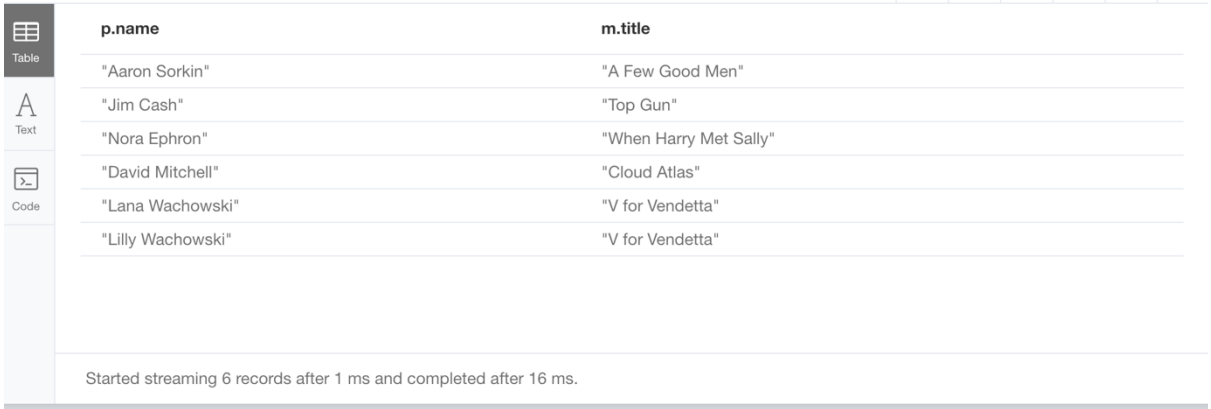


Table | Text | Code

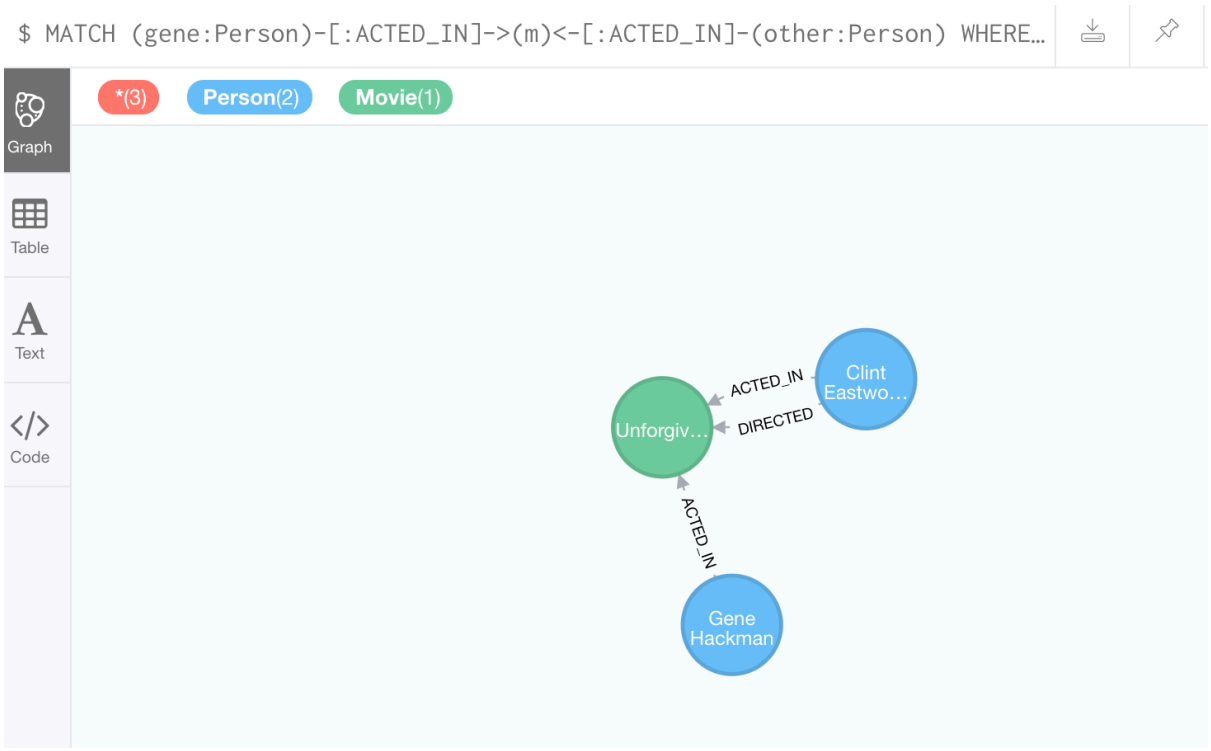
p.name	m.title
"Aaron Sorkin"	"A Few Good Men"
"Jim Cash"	"Top Gun"
"Nora Ephron"	"When Harry Met Sally"
"David Mitchell"	"Cloud Atlas"
"Lana Wachowski"	"V for Vendetta"
"Lilly Wachowski"	"V for Vendetta"

Started streaming 6 records after 1 ms and completed after 16 ms.

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
```

Here is the result:



Testing with list values

If you have a set of values you want to test with, you can place them in a list or you can test with an existing list in the graph.

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.

In this example, we only want to retrieve *Person* nodes of people born in 1965 or 1970:

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

Here is the result:

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

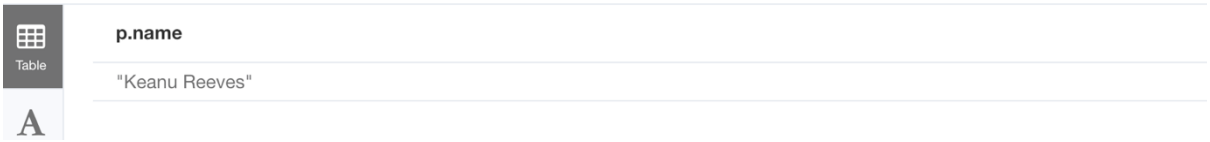
You can also compare a value to an existing list in the graph.

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
```

Here is the result:

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



NOTE	There are a number of syntax elements of Cypher that we have not covered in this training. For example, you can specify `CASE` logic in your conditional testing for your `WHERE` clauses. You can learn more about these syntax elements in the *Neo4j Cypher Manual* and the *Cypher Refcard*.

Exercise 4: Filtering queries using the `WHERE` clause

In the query edit pane of Neo4j Browser, execute the browser command: `:play intro-neo4j-exercises` and follow the instructions for Exercise 4.

Controlling query processing

Now that you have learned how to provide filters for your queries by testing properties, relationships, and patterns using the `WHERE` clause, you will learn some additional Cypher techniques for controlling what the graph engine does during the query.

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
```

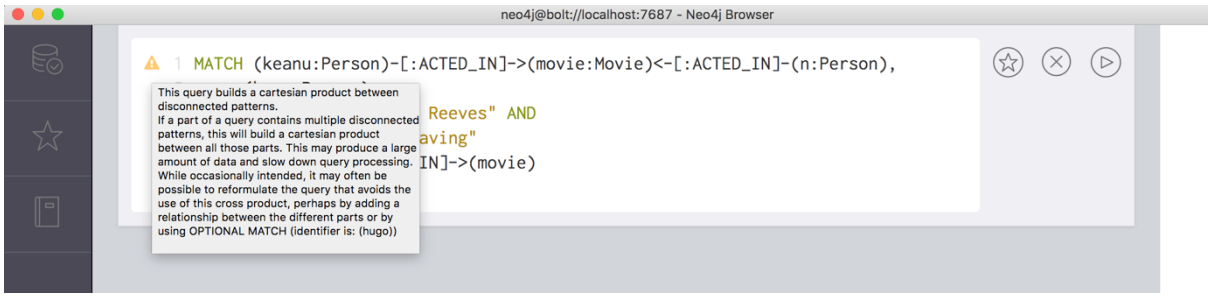
There are, however, some queries where you will need to specify two or more patterns. Multiple patterns are used when a query is complex and cannot be satisfied with a single pattern. This is useful when you are looking for a specific node in the graph and want to connect it to a different node. You will learn about creating nodes and relationships later in this training.

Example 1: Using two MATCH patterns

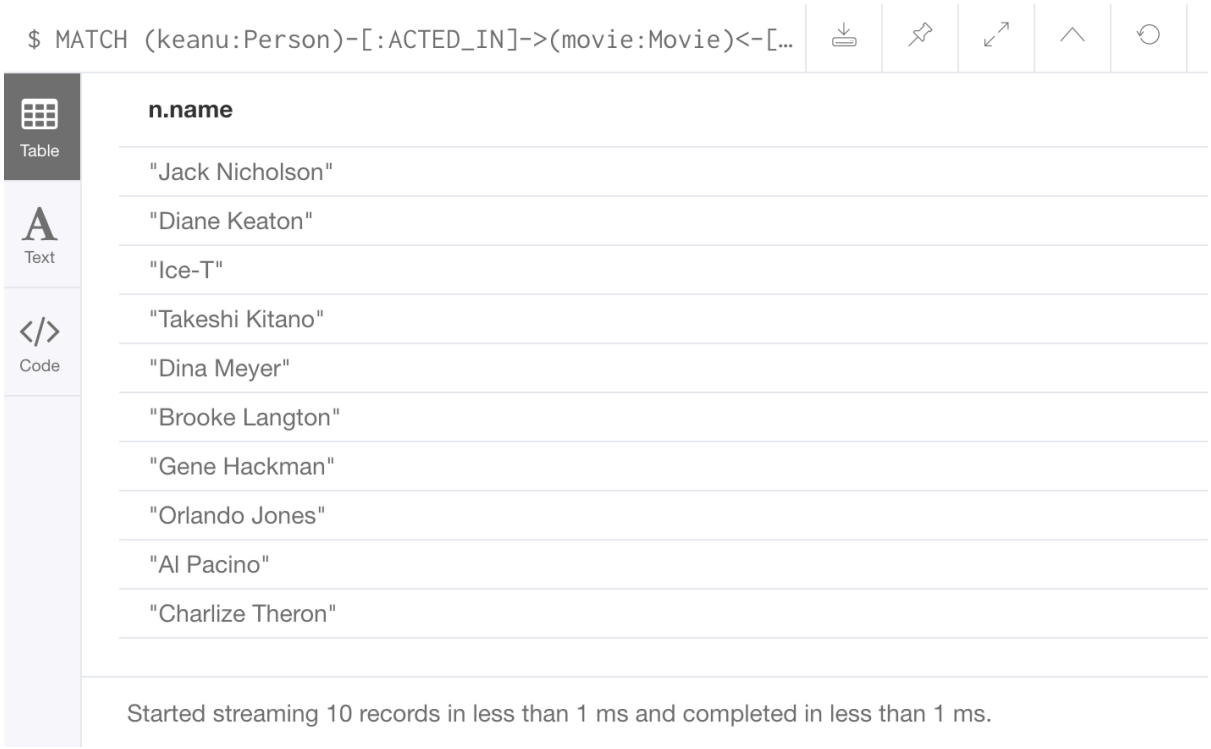
Here are some examples of specifying two paths in a `MATCH` clause. In the first example, we want the actors that worked with *Keanu Reeves* to meet *Hugo Weaving*, who has worked with *Keanu Reeves*. Here we retrieve the actors who acted in the same movies as *Keanu Reeves*, but not when *Hugo Weaving*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

When you perform this type of query, you may see a warning in the query edit pane stating that the pattern represents a cartesian product and may require a lot of resources to perform the query. You should only perform these types of queries if you know the data well and the implications of doing the query.



Here is the result of executing this query:

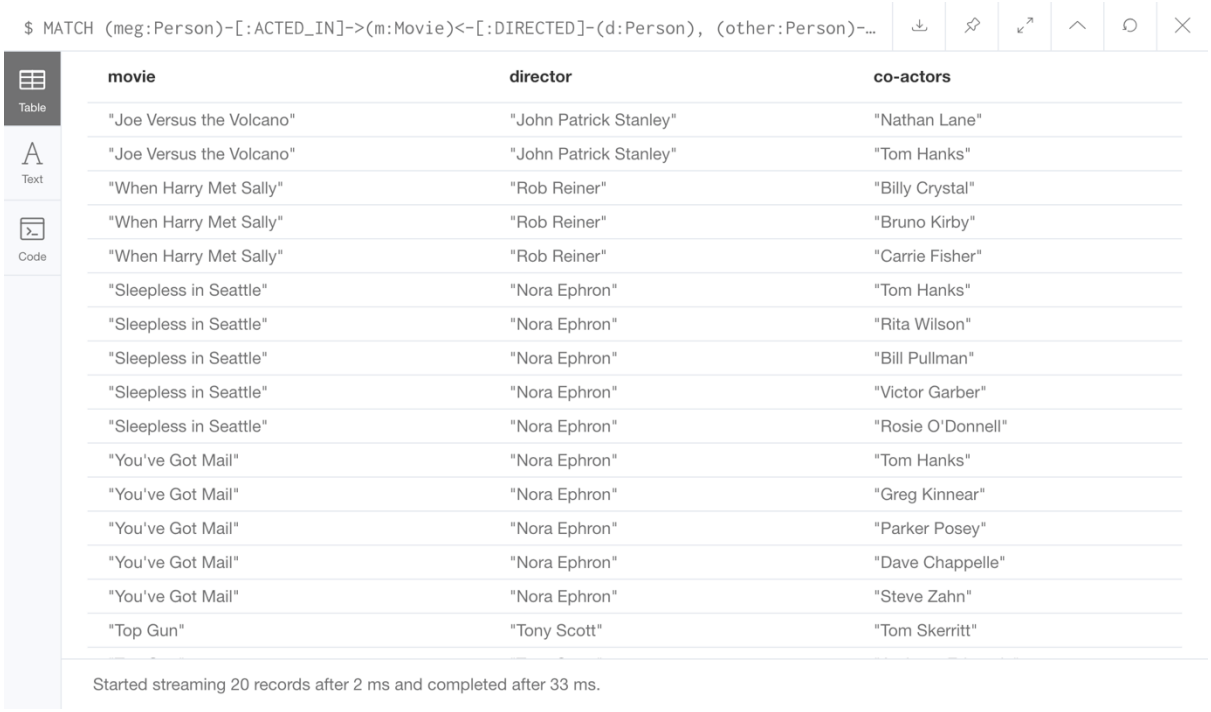


Example 2: Using two MATCH patterns

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`
```

Here is the result returned:



movie	director	co-actors
"Joe Versus the Volcano"	"John Patrick Stanley"	"Nathan Lane"
"Joe Versus the Volcano"	"John Patrick Stanley"	"Tom Hanks"
"When Harry Met Sally"	"Rob Reiner"	"Billy Crystal"
"When Harry Met Sally"	"Rob Reiner"	"Bruno Kirby"
"When Harry Met Sally"	"Rob Reiner"	"Carrie Fisher"
"Sleepless in Seattle"	"Nora Ephron"	"Tom Hanks"
"Sleepless in Seattle"	"Nora Ephron"	"Rita Wilson"
"Sleepless in Seattle"	"Nora Ephron"	"Bill Pullman"
"Sleepless in Seattle"	"Nora Ephron"	"Victor Garber"
"Sleepless in Seattle"	"Nora Ephron"	"Rosie O'Donnell"
"You've Got Mail"	"Nora Ephron"	"Tom Hanks"
"You've Got Mail"	"Nora Ephron"	"Greg Kinnear"
"You've Got Mail"	"Nora Ephron"	"Parker Posey"
"You've Got Mail"	"Nora Ephron"	"Dave Chappelle"
"You've Got Mail"	"Nora Ephron"	"Steve Zahn"
"Top Gun"	"Tony Scott"	"Tom Skerritt"

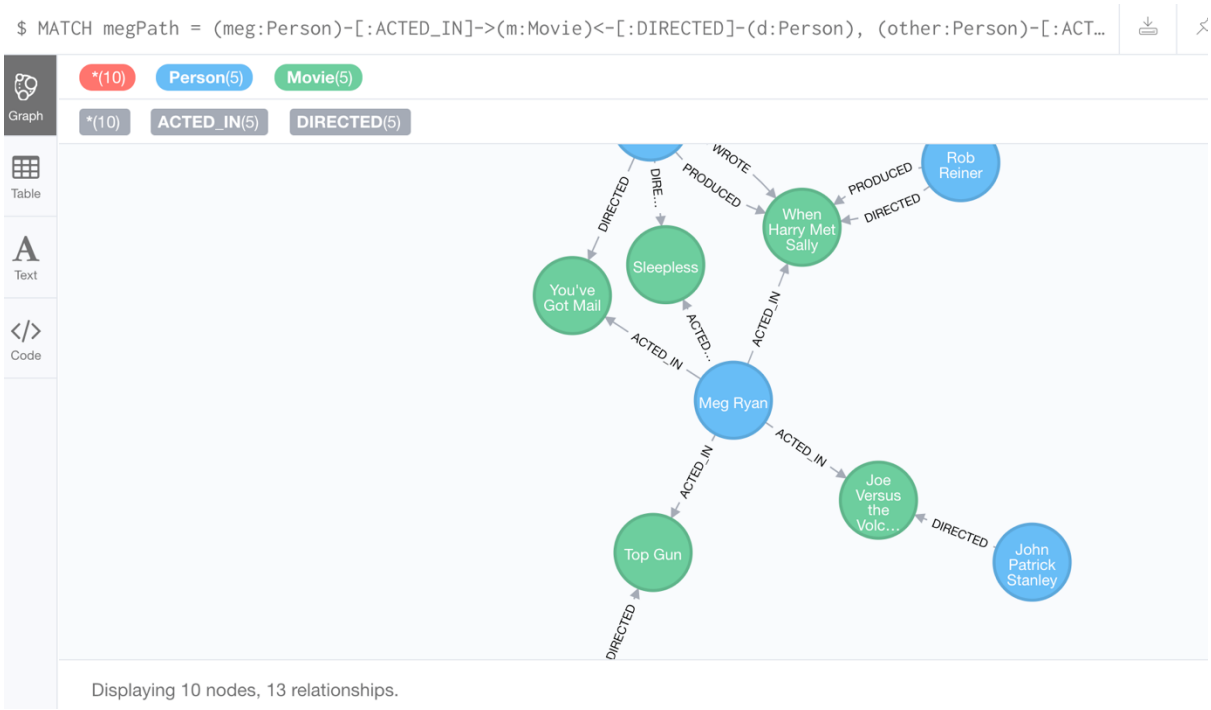
Started streaming 20 records after 2 ms and completed after 33 ms.

Setting path variables

You have previously seen how you can assign a path used in a `MATCH` clause to a variable. This is useful if you want to reuse the path later in the same query or if you want to return the path. So the previous Cypher statement could return the path as follows:

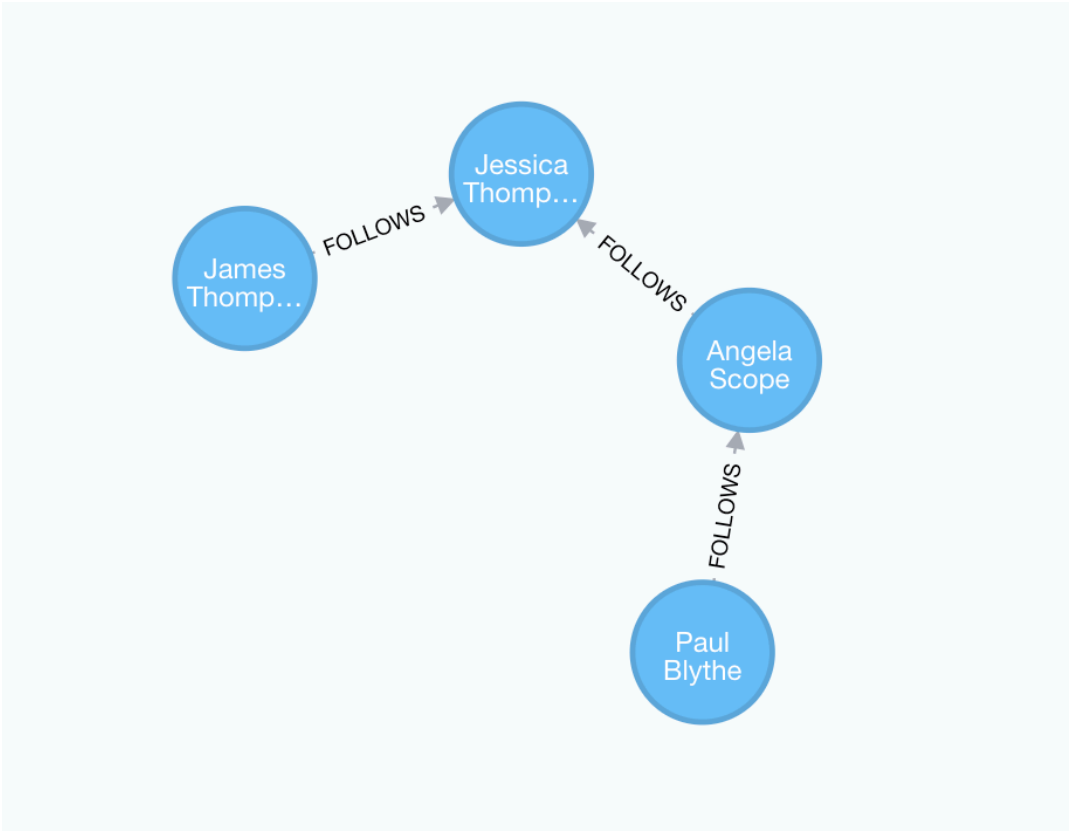
```
MATCH megPath = (meg:Person)-[:ACTED_IN]->(m:Movie)<-
[:DIRECTED]-(d:Person),
      (other:Person)-[:ACTED_IN]->(m)
WHERE meg.name = 'Meg Ryan'
RETURN megPath
```

Here is the result returned:



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
```

Here is the result returned:

If we had specified `[:FOLLOWS*]` rather than `[:FOLLOWS*2]`, 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:

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

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

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)
```

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

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

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)
```

You can learn more about varying paths in the *Patterns* section of the *Neo4j Cypher Manual*.

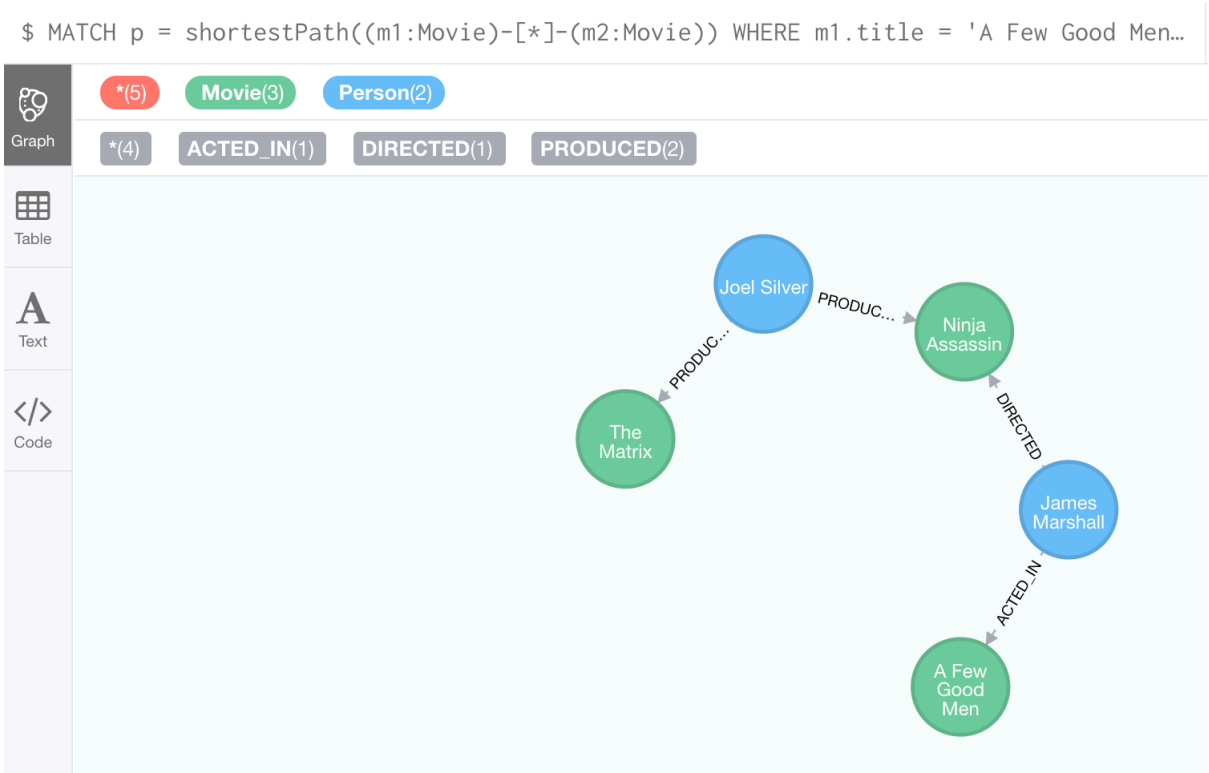
Finding the shortest path

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
```

Here is the result returned:



Notice that the graph engine has traversed many types of relationships to get to the end node.

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. Read the *Graph Algorithms* documentation about the shortest path algorithm.

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.

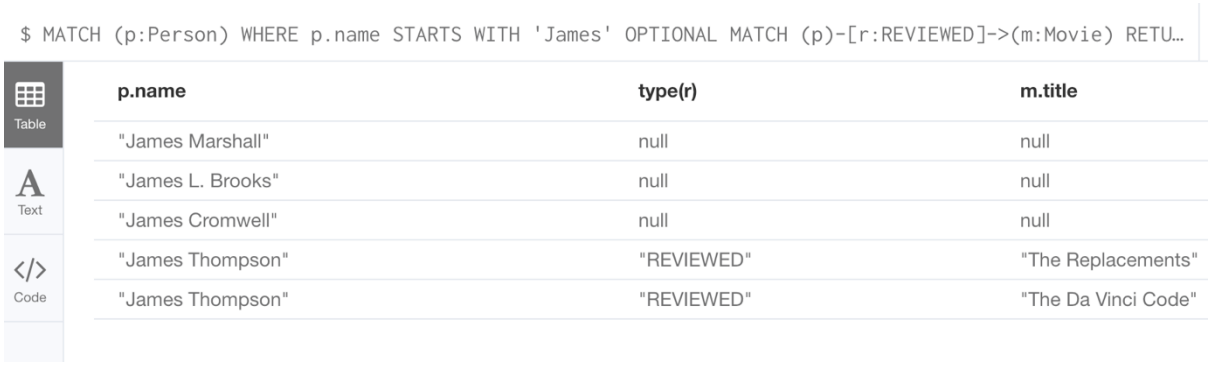
Specifying optional pattern matching

`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
```

Here is the result returned:



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.

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.

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

```
// implicitly groups by a.name and d.name
```

```
MATCH (a)-[:ACTED_IN]->(m)<-[:DIRECTED]-(d)
RETURN a.name, d.name, count(*)
```

With this result returned:

```
$ MATCH (a)-[:ACTED_IN]->(m)<-[:DIRECTED]-(d) RETURN a.name, d.name, count(*)
```

Table | Text | Code

a.name	d.name	count(*)
"Lori Petty"	"Penny Marshall"	1
"Emile Hirsch"	"Lana Wachowski"	1
"Val Kilmer"	"Tony Scott"	1
"Gene Hackman"	"Howard Deutch"	1
"Rick Yune"	"James Marshall"	1
"Audrey Tautou"	"Ron Howard"	1
"Halle Berry"	"Tom Tykwer"	1
"Cuba Gooding Jr."	"James L. Brooks"	1
"Kevin Bacon"	"Rob Reiner"	1
"Tom Hanks"	"Ron Howard"	2
"Laurence Fishburne"	"Lana Wachowski"	3
"Hugo Weaving"	"Lana Wachowski"	4
"Jay Mohr"	"Cameron Crowe"	1
"Hugo Weaving"	"James Marshall"	1
"Philip Seymour Hoffman"	"Mike Nichols"	1
"Werner Herzog"	"Vincent Ward"	1

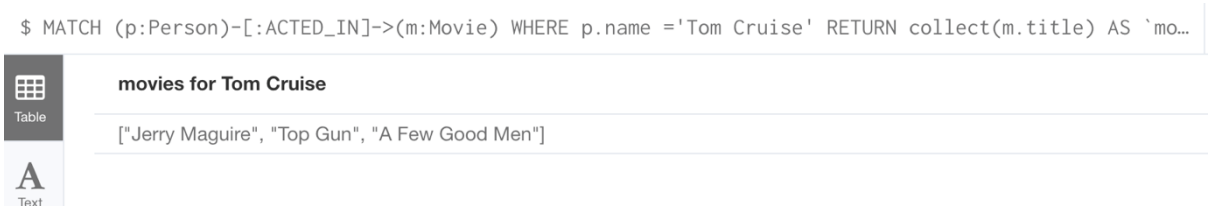
Started streaming 175 records after 8 ms and completed after 8 ms.

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`
```

Here is the result returned:



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

Table | Text

movies for Tom Cruise
["Jerry Maguire", "Top Gun", "A Few Good Men"]

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.

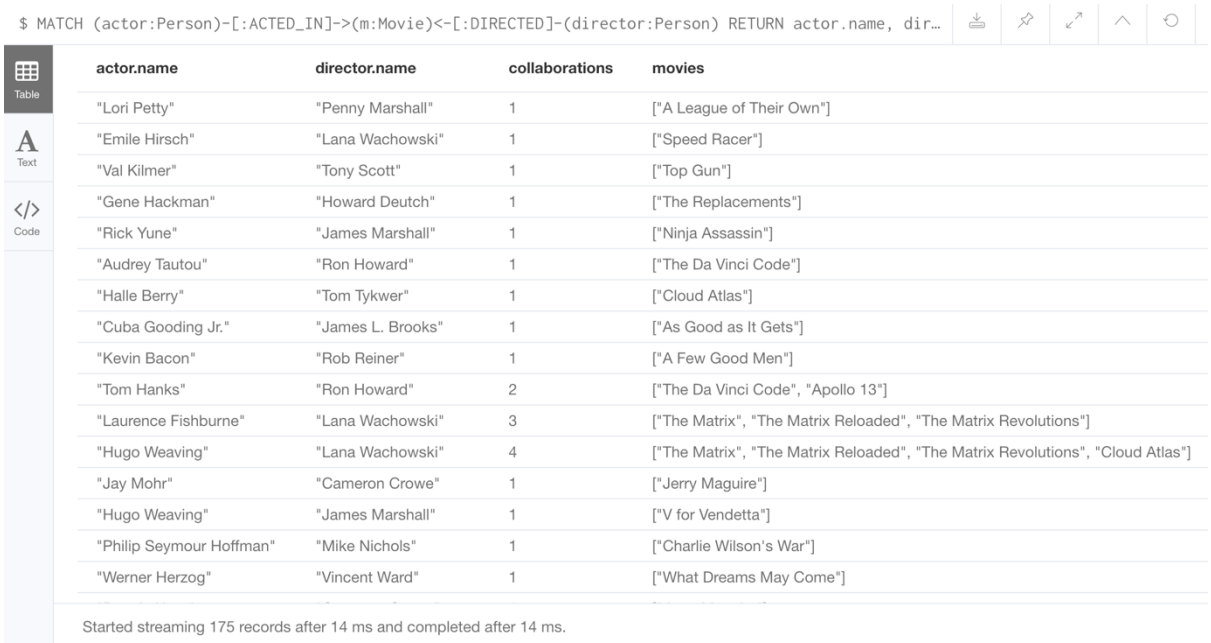
Counting results

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
```

Here is the result returned:



actor.name	director.name	collaborations	movies
"Lori Petty"	"Penny Marshall"	1	["A League of Their Own"]
"Emile Hirsch"	"Lana Wachowski"	1	["Speed Racer"]
"Val Kilmer"	"Tony Scott"	1	["Top Gun"]
"Gene Hackman"	"Howard Deutch"	1	["The Replacements"]
"Rick Yune"	"James Marshall"	1	["Ninja Assassin"]
"Audrey Tautou"	"Ron Howard"	1	["The Da Vinci Code"]
"Halle Berry"	"Tom Tykwer"	1	["Cloud Atlas"]
"Cuba Gooding Jr."	"James L. Brooks"	1	["As Good as It Gets"]
"Kevin Bacon"	"Rob Reiner"	1	["A Few Good Men"]
"Tom Hanks"	"Ron Howard"	2	["The Da Vinci Code", "Apollo 13"]
"Laurence Fishburne"	"Lana Wachowski"	3	["The Matrix", "The Matrix Reloaded", "The Matrix Revolutions"]
"Hugo Weaving"	"Lana Wachowski"	4	["The Matrix", "The Matrix Reloaded", "The Matrix Revolutions", "Cloud Atlas"]
"Jay Mohr"	"Cameron Crowe"	1	["Jerry Maguire"]
"Hugo Weaving"	"James Marshall"	1	["V for Vendetta"]
"Philip Seymour Hoffman"	"Mike Nichols"	1	["Charlie Wilson's War"]
"Werner Herzog"	"Vincent Ward"	1	["What Dreams May Come"]

Started streaming 175 records after 14 ms and completed after 14 ms.

There are more aggregating functions such as `min()` or `max()` that you can also use in your queries. These are described in the *Aggregating Functions* section of the *Neo4j Cypher Manual*.

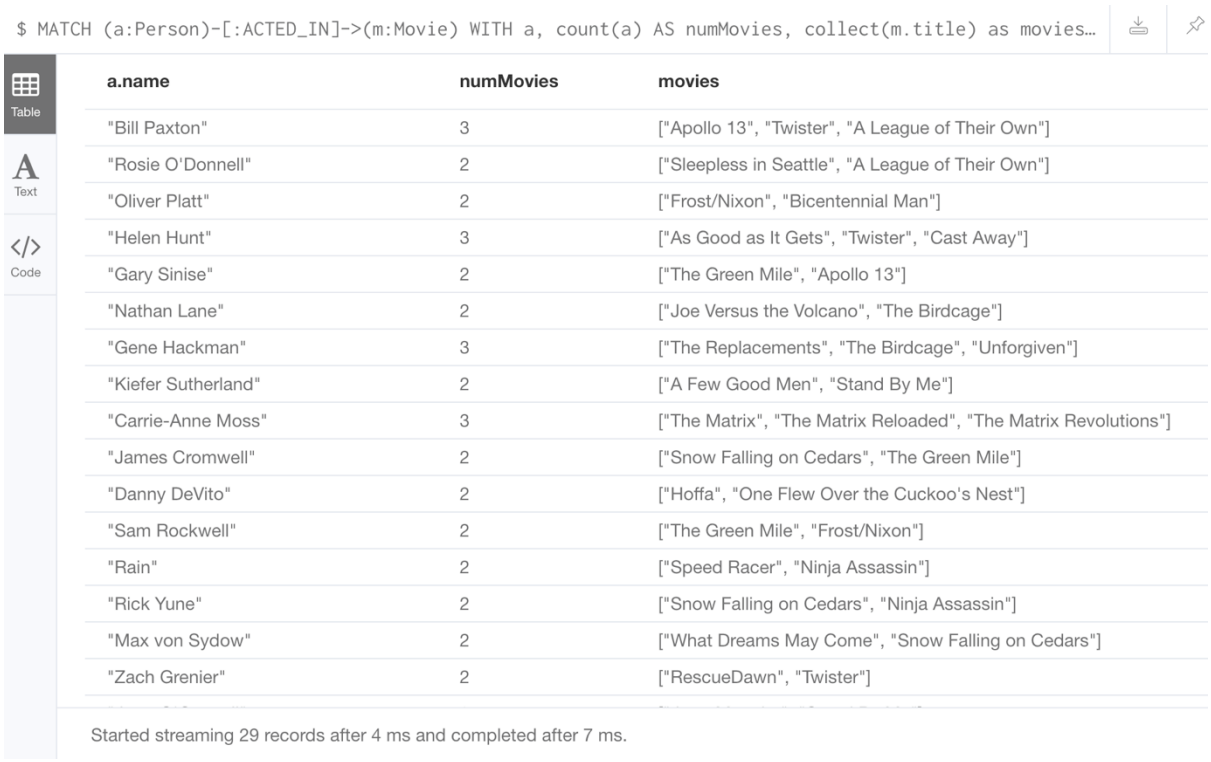
Additional processing using `WITH`

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
```

Here is the result returned:



a.name	numMovies	movies
"Bill Paxton"	3	["Apollo 13", "Twister", "A League of Their Own"]
"Rosie O'Donnell"	2	["Sleepless in Seattle", "A League of Their Own"]
"Oliver Platt"	2	["Frost/Nixon", "Bicentennial Man"]
"Helen Hunt"	3	["As Good as It Gets", "Twister", "Cast Away"]
"Gary Sinise"	2	["The Green Mile", "Apollo 13"]
"Nathan Lane"	2	["Joe Versus the Volcano", "The Birdcage"]
"Gene Hackman"	3	["The Replacements", "The Birdcage", "Unforgiven"]
"Kiefer Sutherland"	2	["A Few Good Men", "Stand By Me"]
"Carrie-Anne Moss"	3	["The Matrix", "The Matrix Reloaded", "The Matrix Revolutions"]
"James Cromwell"	2	["Snow Falling on Cedars", "The Green Mile"]
"Danny DeVito"	2	["Hoffa", "One Flew Over the Cuckoo's Nest"]
"Sam Rockwell"	2	["The Green Mile", "Frost/Nixon"]
"Rain"	2	["Speed Racer", "Ninja Assassin"]
"Rick Yune"	2	["Snow Falling on Cedars", "Ninja Assassin"]
"Max von Sydow"	2	["What Dreams May Come", "Snow Falling on Cedars"]
"Zach Grenier"	2	["RescueDawn", "Twister"]

Started streaming 29 records after 4 ms and completed after 7 ms.

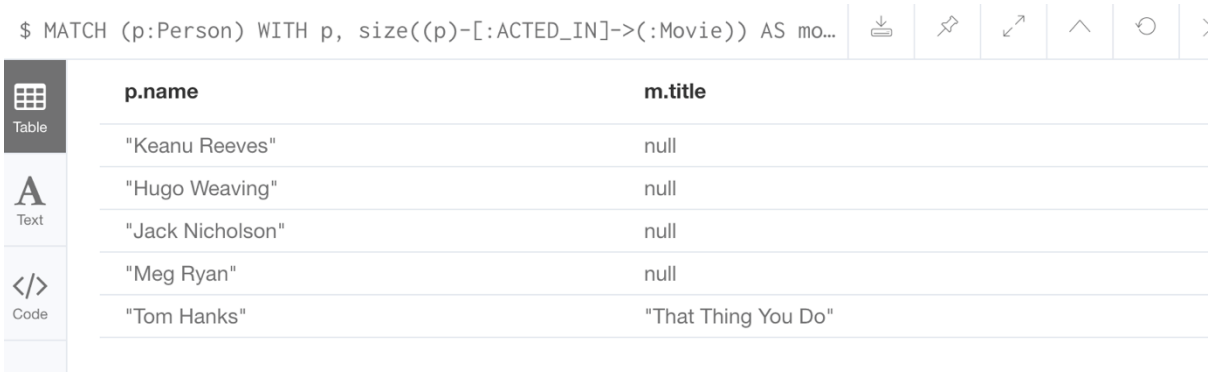
When you use the `WITH` clause, you specify the variables from the previous part of the query you want to pass on to the next part of the query. In this example, the variable *a* is specified to be passed on in the query, but *m* is not. Since *m* is not specified to be passed on, *m* will not be available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

NOTE	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
```

Here is the result returned:



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.

Exercise 5: Controlling query processing

In the query edit pane of Neo4j Browser, execute the browser command: `:play intro-neo4j-exercises` and follow the instructions for Exercise 5.

Controlling how results are returned

Next, you will learn some additional Cypher techniques for controlling how results are returned from a query.

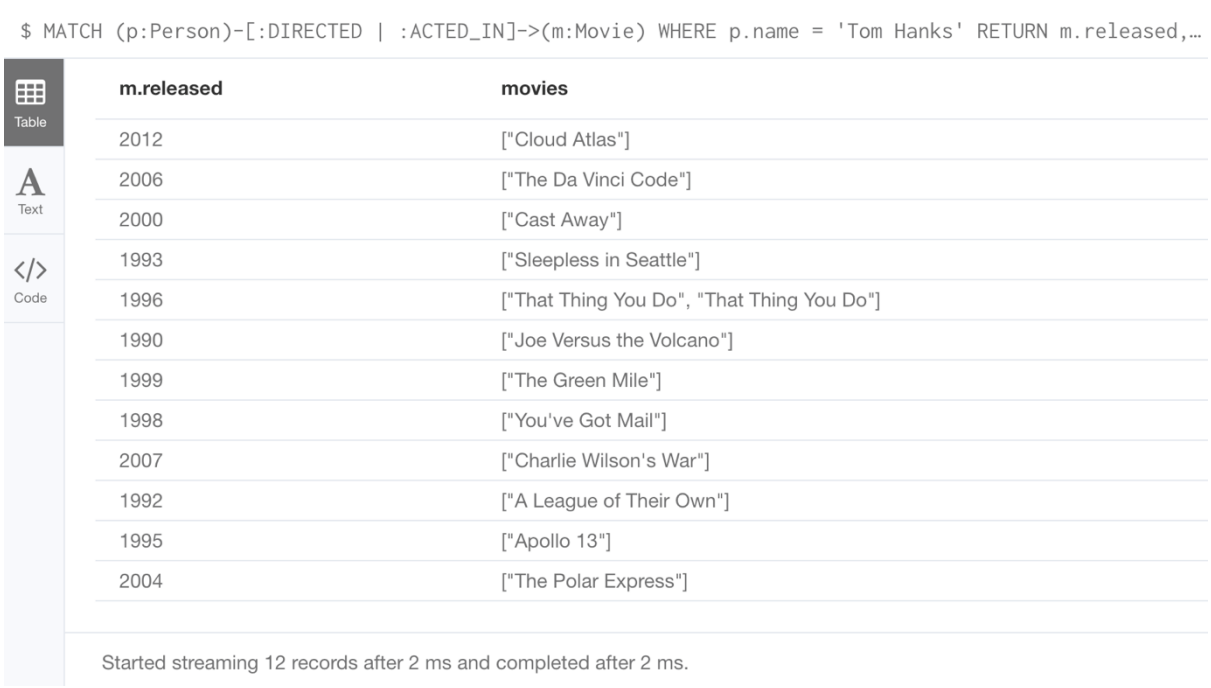
Eliminating duplication

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'
RETURN m.released, collect(m.title) AS movies
```

Here is the result returned:



m.released	movies
2012	["Cloud Atlas"]
2006	["The Da Vinci Code"]
2000	["Cast Away"]
1993	["Sleepless in Seattle"]
1996	["That Thing You Do", "That Thing You Do"]
1990	["Joe Versus the Volcano"]
1999	["The Green Mile"]
1998	["You've Got Mail"]
2007	["Charlie Wilson's War"]
1992	["A League of Their Own"]
1995	["Apollo 13"]
2004	["The Polar Express"]

Started streaming 12 records after 2 ms and completed after 2 ms.

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
```

Here is the result returned:

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

Table | Text | Code

m.released	movies
2012	["Cloud Atlas"]
2006	["The Da Vinci Code"]
2000	["Cast Away"]
1993	["Sleepless in Seattle"]
1996	["That Thing You Do"]
1990	["Joe Versus the Volcano"]
1999	["The Green Mile"]
1998	["You've Got Mail"]
2007	["Charlie Wilson's War"]
1992	["A League of Their Own"]
1995	["Apollo 13"]
2004	["The Polar Express"]

Started streaming 12 records after 1 ms and completed after 1 ms.

Using WITH and DISTINCT to eliminate duplication

Another way that you can avoid duplication is to with `WITH` and `DISTINCT` together as follows:

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

Here is the result returned:

```
$ MATCH (p:Person)-[:DIRECTED | :ACTED_IN]->(m:Movie) WHERE p.name = 'Tom Hanks' WITH distinct m RE…
```

Table | Text | Code

m.released	m.title
2004	"The Polar Express"
1995	"Apollo 13"
1990	"Joe Versus the Volcano"
1992	"A League of Their Own"
1999	"The Green Mile"
1996	"That Thing You Do"
1998	"You've Got Mail"
2000	"Cast Away"
2012	"Cloud Atlas"
1993	"Sleepless in Seattle"
2007	"Charlie Wilson's War"
2006	"The Da Vinci Code"

Started streaming 12 records after 1 ms and completed after 1 ms.

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.

In this example, we specify that the release date of the movies for *Tom Hanks* will be returned in descending order.

```
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
```

Here is the result returned:

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

Table
Text
Code

m.released	movies
2012	["Cloud Atlas"]
2007	["Charlie Wilson's War"]
2006	["The Da Vinci Code"]
2004	["The Polar Express"]
2000	["Cast Away"]
1999	["The Green Mile"]
1998	["You've Got Mail"]
1996	["That Thing You Do"]
1995	["Apollo 13"]
1993	["Sleepless in Seattle"]
1992	["A League of Their Own"]
1990	["Joe Versus the Volcano"]

Started streaming 12 records after 2 ms and completed after 2 ms.

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.

Suppose you want to see the titles of the ten most recently released movies. You could do so as follows where you limit the number of results using the `LIMIT` keyword as follows:

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

Here is the result returned:

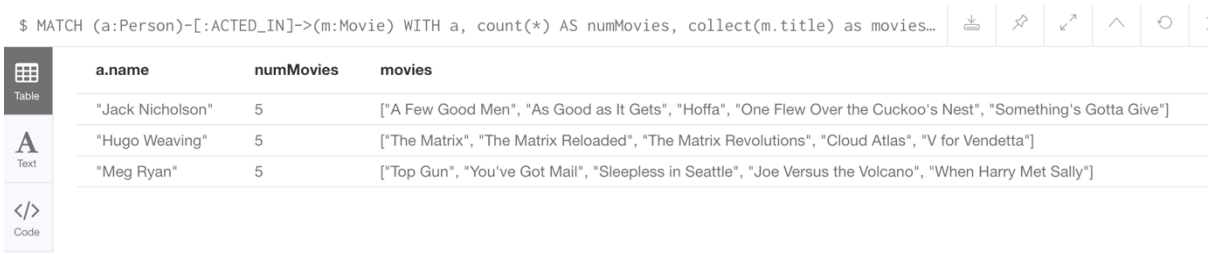
Controlling the number of results using WITH

Previously, you saw how you can use the `WITH` clause to perform some intermediate processing during a query. You can use the `WITH` clause to limit the number of results.

In this example, we count the number of movies during the query and we return the results once we have reached 5 movies:

```
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
```

Here is the result returned:



Exercise 6: Controlling results returned

In the query edit pane of Neo4j Browser, execute the browser command: `:play intro-neo4j-exercises` and follow the instructions for Exercise 6.

Working with Cypher data

Thus far, you have specified both string and numeric types in your Cypher queries. You have also learned that nodes and relationships can have properties, whose values are structured like JSON objects. You have also learned that the `collect()` function can create lists of values or objects where a list is comma-separated and you can use the `IN` keyword to search for a value in a list. Next, you will learn more about working with lists and dates in Cypher.

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)
```

Here is the result returned:

```
$ MATCH (a:Person)-[:ACTED_IN]->(m:Movie) WITH m, count(m) AS numCast, collect(a.nam…
```

Table | Text | Code

m.title	cast	numCast
"The Polar Express"	["Tom Hanks"]	1
"Cast Away"	["Helen Hunt", "Tom Hanks"]	2
"One Flew Over the Cuckoo's Nest"	["Jack Nicholson", "Danny DeVito"]	2
"Bicentennial Man"	["Robin Williams", "Oliver Platt"]	2
"Joe Versus the Volcano"	["Nathan Lane", "Meg Ryan", "Tom Hanks"]	3
"The Devil's Advocate"	["Keanu Reeves", "Al Pacino", "Charlize Theron"]	3
"The Birdcage"	["Nathan Lane", "Gene Hackman", "Robin Williams"]	3

You can read more about working with lists in the *List Functions* section of the *Neo4j Cypher Manual*.

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
```

Here is the result returned:

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

Table | Text | </>

list	row
[1, 2, 3]	1
[1, 2, 3]	2
[1, 2, 3]	3

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.

NOTE	The `UNWIND` clause is frequently used when importing data into a graph.

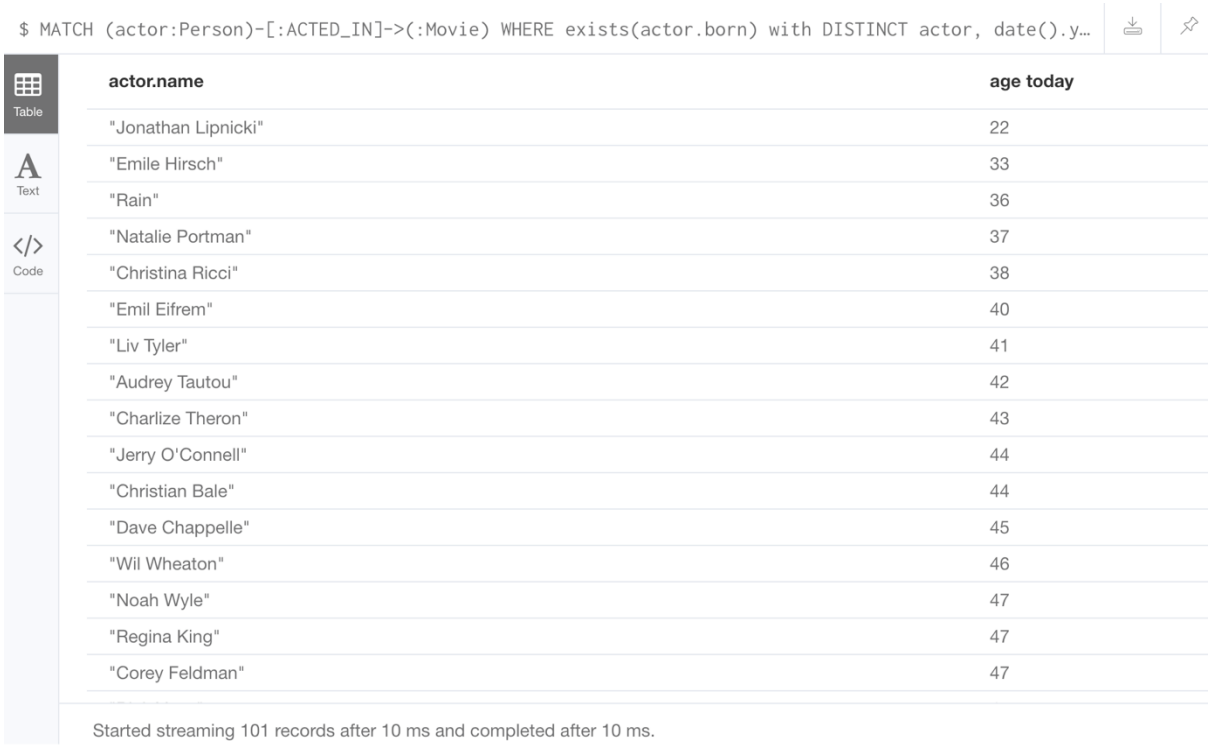
Dates

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).

Here is example Cypher to retrieve all actors from the graph, and if they have a value for *born*, calculate the *age* value.

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

Here is the result returned:



Consult the *Neo4j Cypher Manual* for more information about the built-in functions available for working with data of all types:

- Predicate
- Scalar
- List
- Mathematical
- String
- Temporal
- Spatial

Exercise 7: Working with Cypher data

In the query edit pane of Neo4j Browser, execute the browser command: `:play intro-neo4j-exercises` and follow the instructions for Exercise 7.

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:

Add a node to the graph of type *Movie* with the *title Batman Begins*. This node can be retrieved using the title. A set of nodes with the label *Movie* can also be retrieved which will contain this node:

```
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'})
```

Add a node 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. The variable *m* can be used for later processing after the `CREATE` clause:

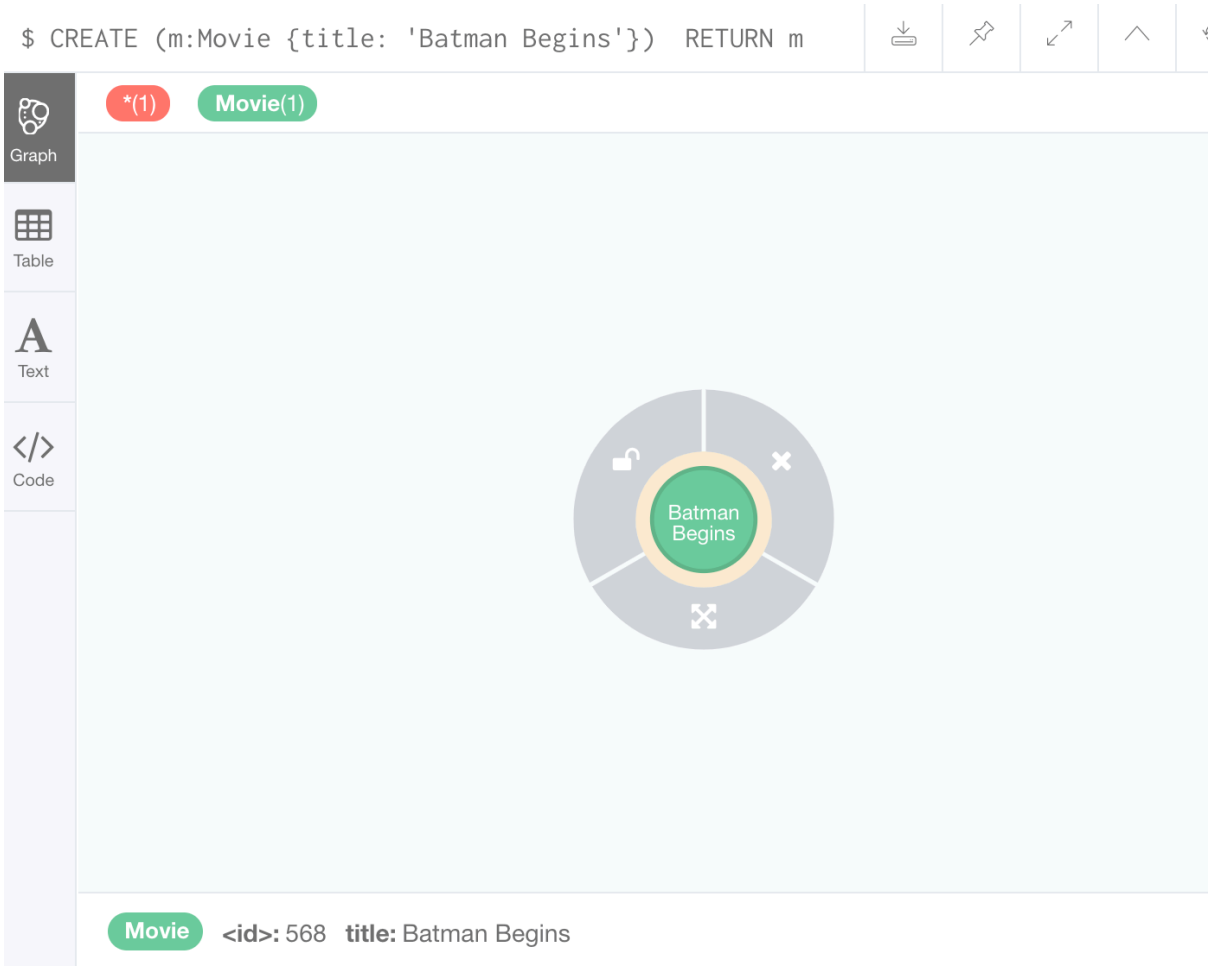
```
CREATE (m:Movie:Action {title: 'Batman Begins'})
```

Add a node 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. Here we return the title of the node:

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

Here is what you see in Neo4j Browser when you create a node for the movie, *Batman Begins* where we have selected the node returned to view its properties.:



When the graph engine creates a node, it automatically assigns a read-only, unique ID to the node. Here we see that the *id* of the node is *568*. This is not a property of a node, but rather an internal value.

After you have created a node, you can add more properties or labels to it and most importantly, connect it to another node.

Creating multiple nodes

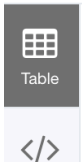
You can create multiple nodes by simply separating the nodes specified with commas, or by specifying multiple CREATE statements.

Here is an example, where we create some *Person* nodes that will represent some of the people associated with the movie *Batman Begins*:

```
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})
```

Here is the result of running this Cypher statement:

```
$ CREATE (:Person {name: 'Michael Caine', born: 1933}), (:Person {name: 'Liam Nees…
```



Added 4 labels, created 4 nodes, set 8 properties, completed after 1 ms.

NOTE

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

You will learn about merging data later in this module. Constraints are configured globally for a graph and are covered later in this training.

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.

Here is the simplified syntax for adding labels to a node:

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

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

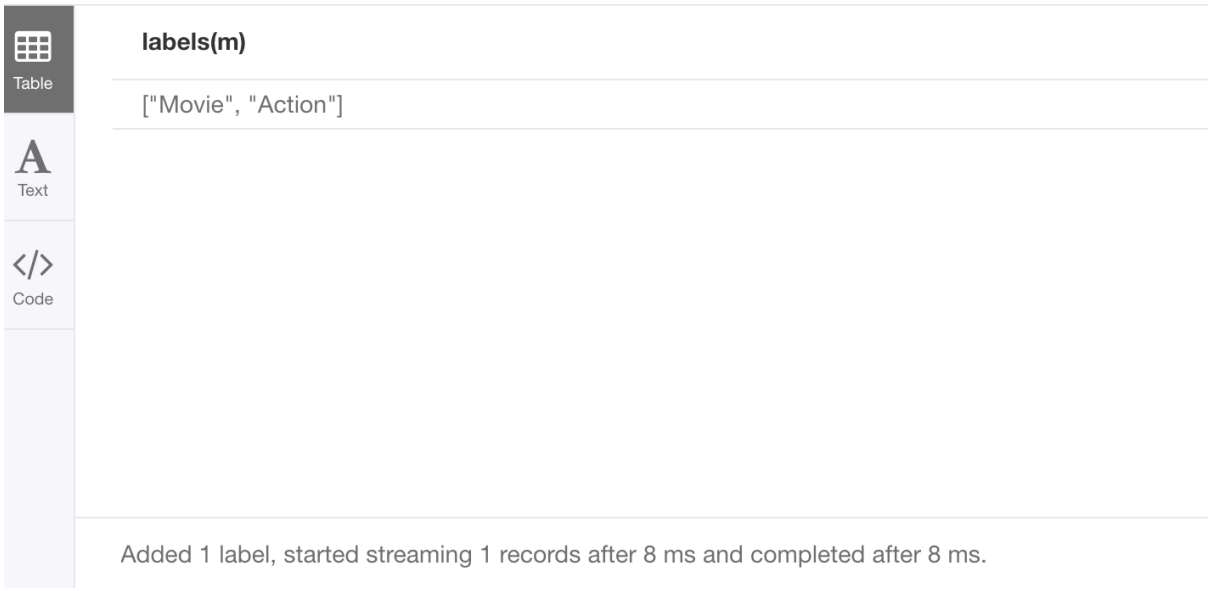
Here is an example where we add the *Action* label to the node that has a label, *Movie*:

```
MATCH (m:Movie)
```

```
WHERE m.title = 'Batman Begins'
SET m:Action
RETURN labels(m)
```

Assuming that we have previously created the node for the movie, here is the result of running this Cypher statement:

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



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

Removing labels from a node

Perhaps your data model has changed or the underlying data for a node has changed so that the label for a node is no longer useful or valid.

Here is the simplified syntax for removing labels from a node:

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

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

Here is an example where we remove the *Action* label from the node that has a labels, *Movie* and *Action*:

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

```
RETURN labels(m)
```

Assuming that we have previously created the node for the movie, here is the result of running this Cypher statement:

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}
```

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 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 *propertyNnameX* 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
```

Assuming that we have previously created the node for the movie, here is the result of running this Cypher statement:

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
```

Here is the result of running this Cypher statement:

```
$ MATCH (m:Movie) WHERE m.title = 'Batman Begins' SET m = {title: 'Batman Begins',  released: 200…
```

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.

```
$ call db.propertyKeys()
```

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
```

Here is the result:

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:

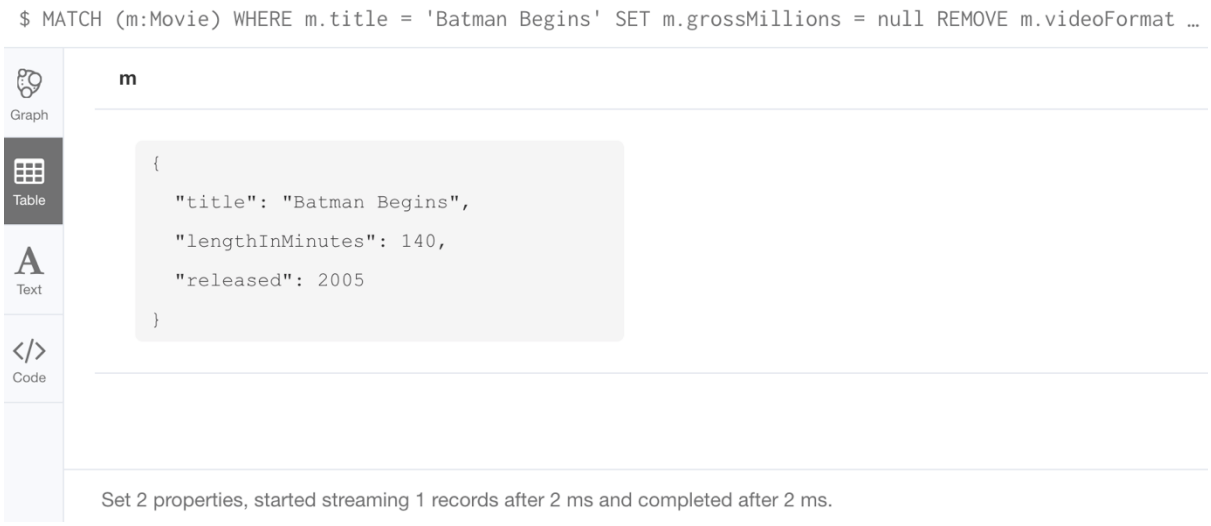
```
REMOVE x.propertyName
SET x.propertyName = null
```

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
```

Assuming that we have previously created the node for the movie with the these properties, here is the result of running this Cypher statement where we remove each property a different way. One way we remove the property using the `SET` clause to set the property to null. And in another way, we use the `REMOVE` clause.



Exercise 8: Creating Nodes

In the query edit pane of Neo4j Browser, execute the browser command: `:play intro-neo4j-exercises` and follow the instructions for Exercise 8.

Creating relationships

As you have learned in the previous exercises where you query the graph, you often query using connections between nodes. 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)
```

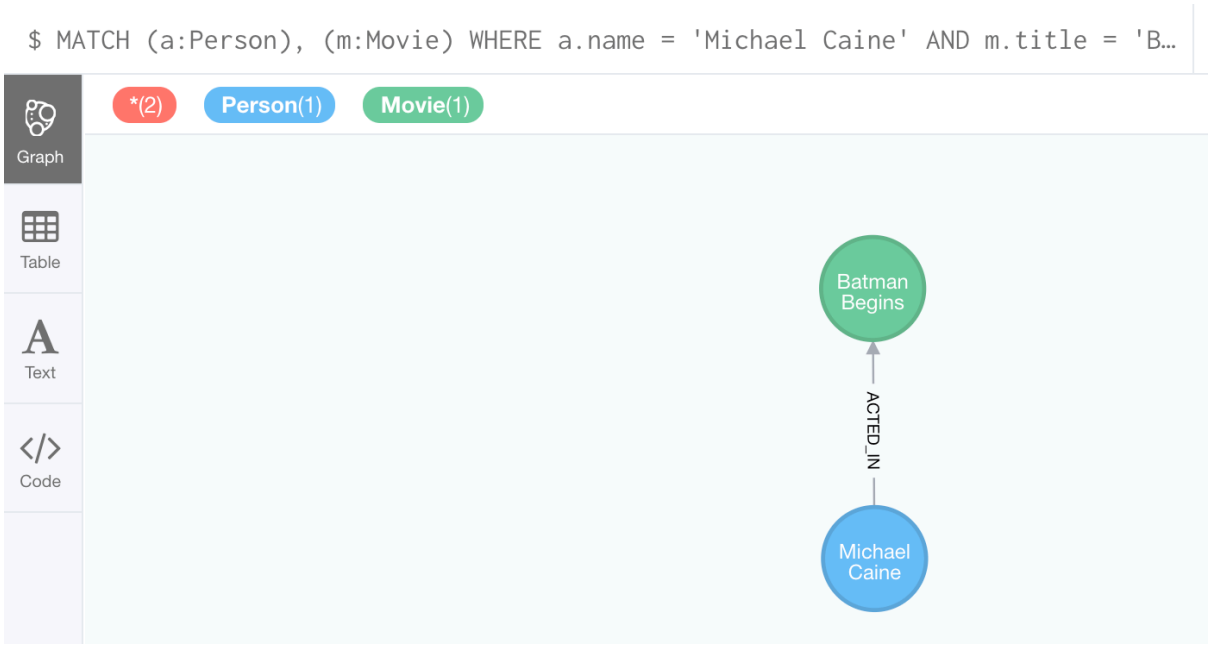
When you create the relationship, it must have direction. You can query nodes for a relationship in either direction, but you must create the relationship with a direction. An exception to this is when you create a node using `MERGE` that you will learn about later in this module.

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
```

Here is the result of running this Cypher statement:



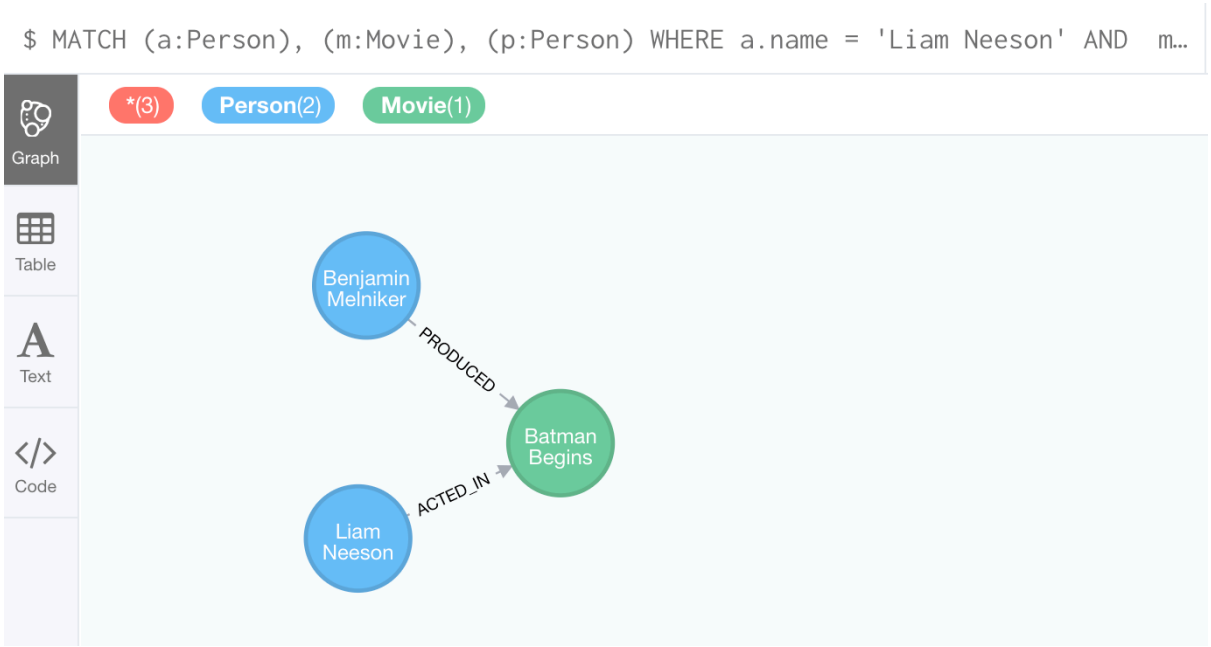
NOTE	Before you run these Cypher statements, you may see a warning in Neo4j Browser that you are creating a query that is a cartesian product that could potentially be a performance issue. You will see this warning if you have no unique constraint on the lookup keys. You will learn about uniqueness constraints 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
```

Here is the result of running this Cypher statement:



NOTE	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

You can add properties to a relationship, just as you add properties to a node. You use the `SET` clause to do so.

Here is the simplified syntax for adding properties to a relationship referenced by the variable r:

```
SET r.propertyName = value
```

```
SET r.propertyName1 = value1    , r.propertyName2 = value2
SET r = {propertyName1: value1, propertyName2: value2}
SET r += {propertyName1: value1, propertyName2: value2}
```

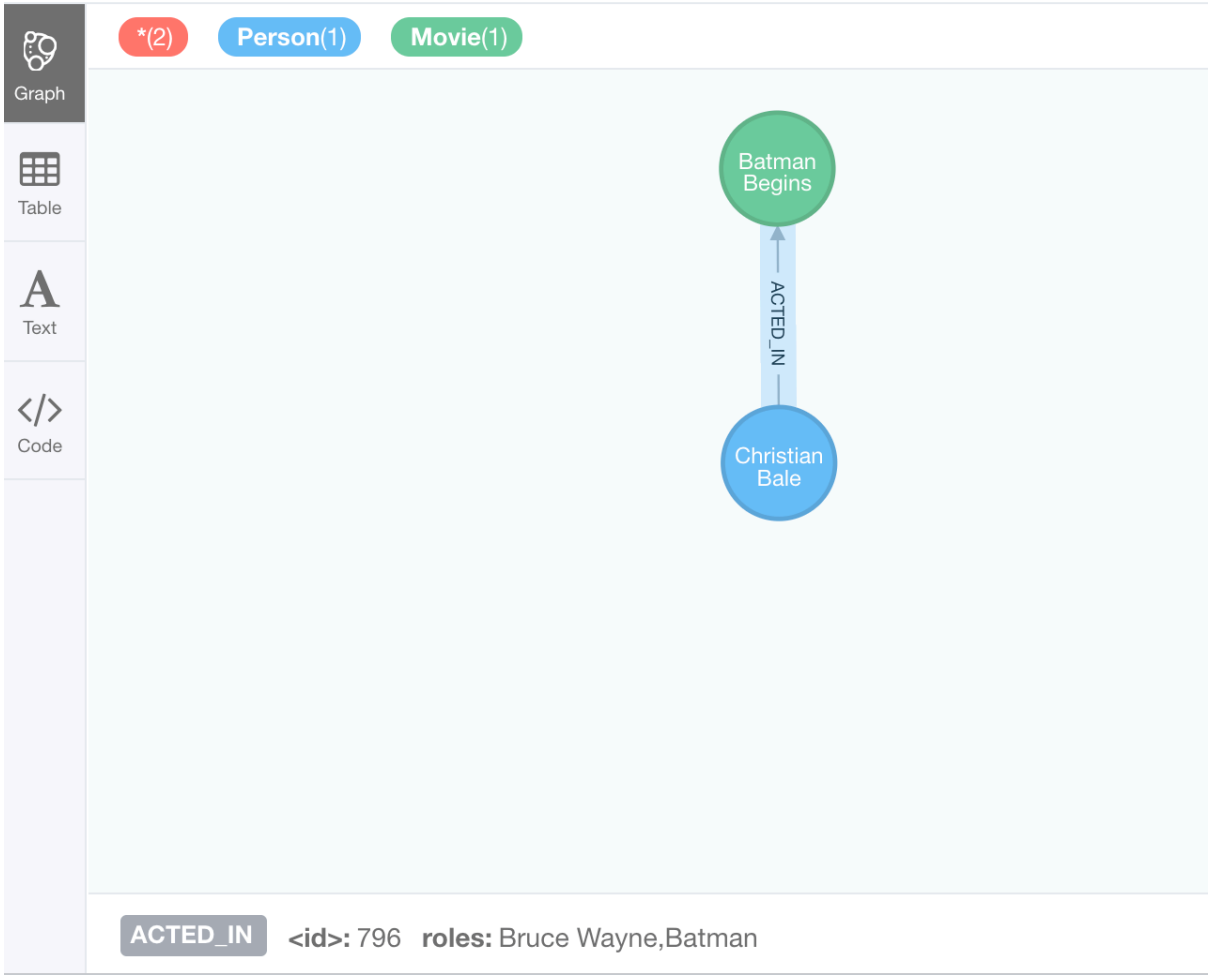
If the property does not exist, it is added to the relationship. If the property exists, its value is updated for the relationship. When specify the JSON-style object for assignment to the relationship using `=`, the object must include all of the properties for the relationship, just as you need to do for nodes. If you use `+=`, you can add or update properties, just as you do for nodes.

Here is an example where we will add the *roles* property to the *ACTED_IN* relationship from *Christian Bale* to *Batman Begins* right after we have created the relationship:

```
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
```

Here is the result of running this Cypher statement:

```
$ MATCH (a:Person), (m:Movie) WHERE a.name = 'Christian Bale' AND m.title
```



The *roles* property is a list so we add it as such. If the relationship had multiple properties, we could have added them as a comma separated list or as an object, like can do for node properties.

You can also add properties to a relationship when the relationship is created. Here is another way to create and add the properties for the relationship:

```
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
```

NOTE	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

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`, just as you do for properties of nodes.

Suppose we have added the *ACTED_IN* relationship between *Christian Bale* and the movie, *Batman Returns* where the *roles* property is added to the relationship. Here is an example to remove the *roles* property, yet keep the *ACTED_IN* relationship:

```
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
```

Here is the result returned. An alternative to `REMOVE rel.roles` would be `SET rel.roles = null`

Getting More Out of Neo4j

Cypher parameters

In a deployed application, you should not **hard code** values in your Cypher statements. You use a variety values when you are testing your Cypher statements. But you don't want to change the Cypher statement every time you test. In addition, you typically include Cypher statements in an application where parameters are passed in to the Cypher statement before it executes. For these scenarios, you should parameterize values in your Cypher statements.

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'
```

NOTE	You can even specify a Cypher expression to the right of `=>` to set the value of the parameter.

Here is the result of executing the `:param` command:

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

```
{
  "actorName": "Tom Hanks"
}
```

See `:help param` for usage of the `:param` command.

Successfully set your parameters.

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.

After the *actorName* parameter is set, you can run the query that uses the parameter:

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

Table | Text | Code

m.released	m.title
2012	"Cloud Atlas"
2007	"Charlie Wilson's War"
2006	"The Da Vinci Code"
2004	"The Polar Express"
2000	"Cast Away"
1999	"The Green Mile"
1998	"You've Got Mail"
1996	"That Thing You Do"
1995	"Apollo 13"
1994	"Forrest Gump"
1993	"Sleepless in Seattle"
1992	"A League of Their Own"
1990	"Joe Versus the Volcano"

Subsequently, you need only change the value of the parameter and not the Cypher statement to test with different values.

After we have changed the *actorName* parameter to ‘Tom Cruise’, we get a different result with the same Cypher query:

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

m.released	m.title
2000	"Jerry Maguire"
1992	"A Few Good Men"
1986	"Top Gun"

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'}
```

With the result:

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

```
{
  "actorName": "Tom Cruise",
  "movieName": "Top Gun"
}
```

See `:help param` for usage of the `:param` command.

Successfully set your parameters.

If you want to remove an existing parameter from your session, you do by 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`:

```
$ :params
```

```
{
  "actorName": "Tom Cruise",
  "movieName": "Top Gun"
}
```

See `:help param` for usage of the `:param` command.

Exercise 12: Using Cypher parameters

In the query edit pane of Neo4j Browser, execute the browser command: `:play intro-neo4j-exercises` and follow the instructions for Exercise 12.

Analyzing Cypher execution

The *Movie* graph that you have been using during training is a very small graph. As you start working with large datasets, it will be important to not only add appropriate indexes to your graph, but also write Cypher statements that execute as efficiently as possible.

There are two Cypher keywords you can prefix a Cypher statement with to analyze a query:

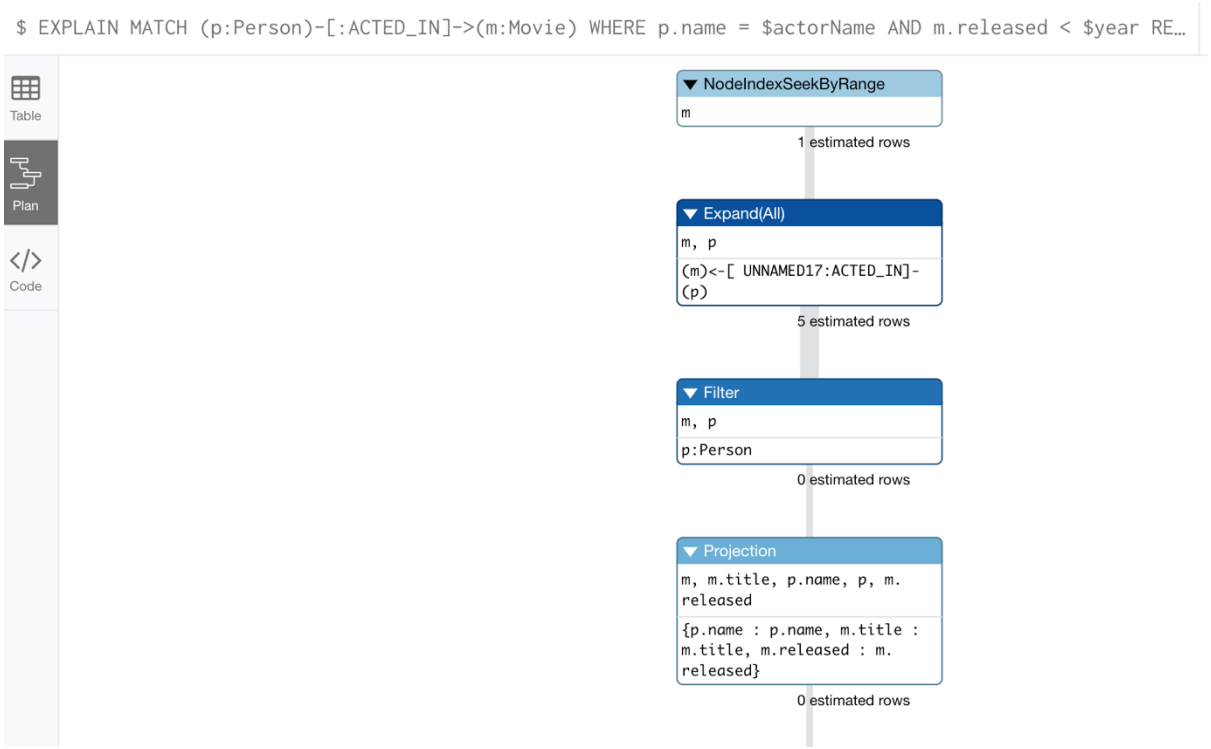
- `EXPLAIN` provides estimates of the graph engine processing that will occur, but does not execute the Cypher statement.
- `PROFILE` provides real profiling information for what has occurred in the graph engine during the query and executes the Cypher statement.

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
```

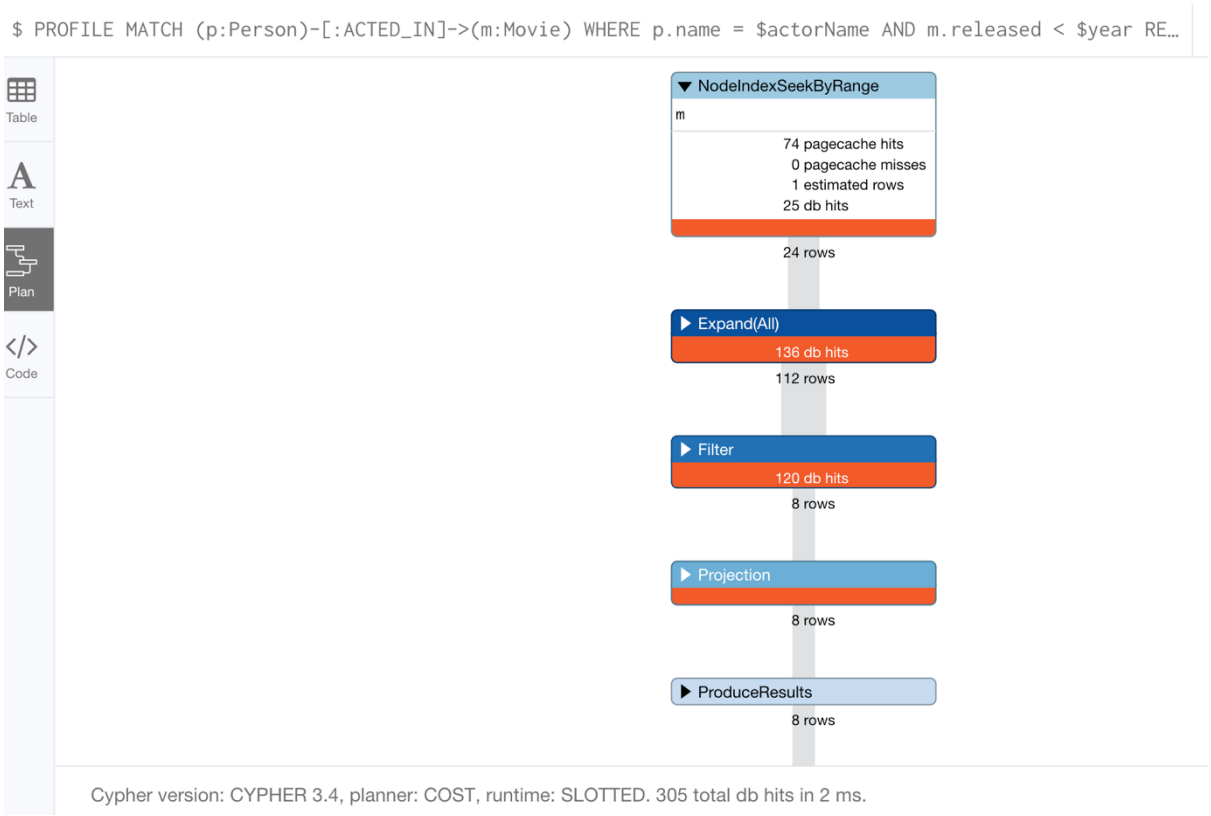
Here is the query plan returned:



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.

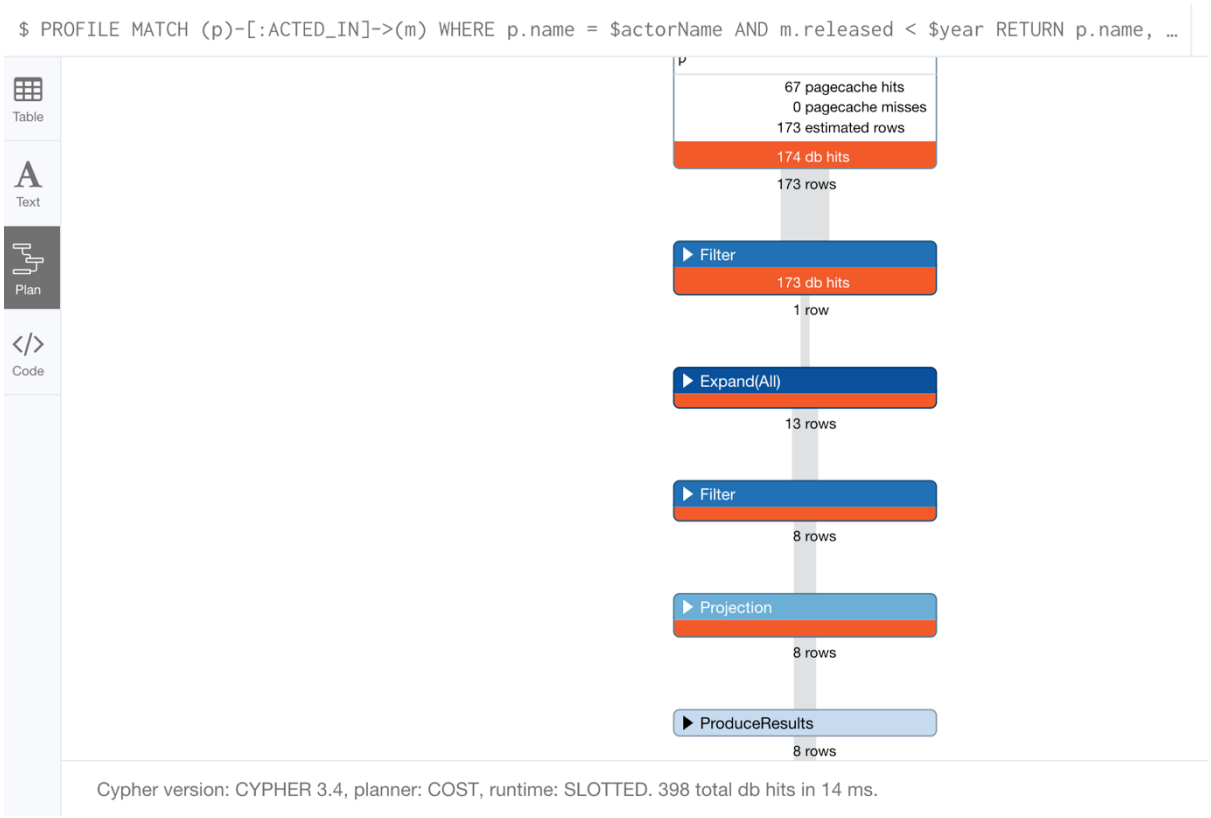
For a better metric for analyzing how the Cypher statement will run you use the `PROFILE` keyword which runs the Cypher statement and gives you run-time performance metrics.

Here is the result returned using `PROFILE` for this Cypher statement:



Here we 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.

For example, if we were to change the Cypher statement so that the node labels are not specified, we see these metrics when we profile:



Here we see more db hits which makes sense because all nodes need to be scanned for perform this query.

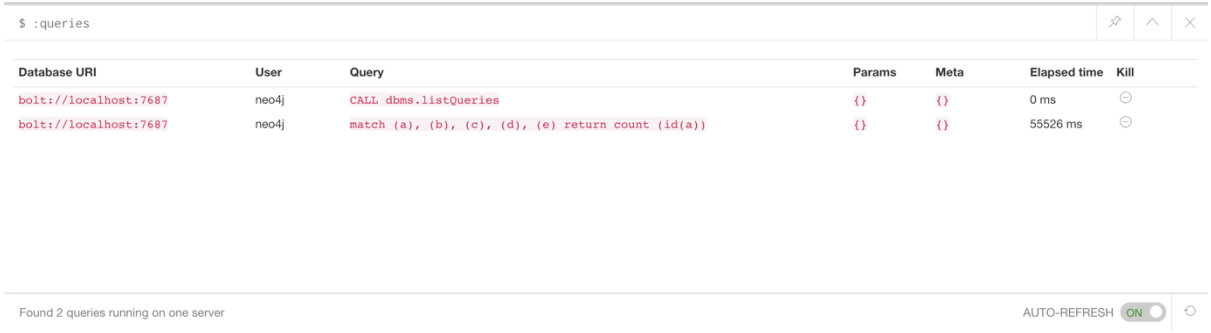
Monitoring queries

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))`

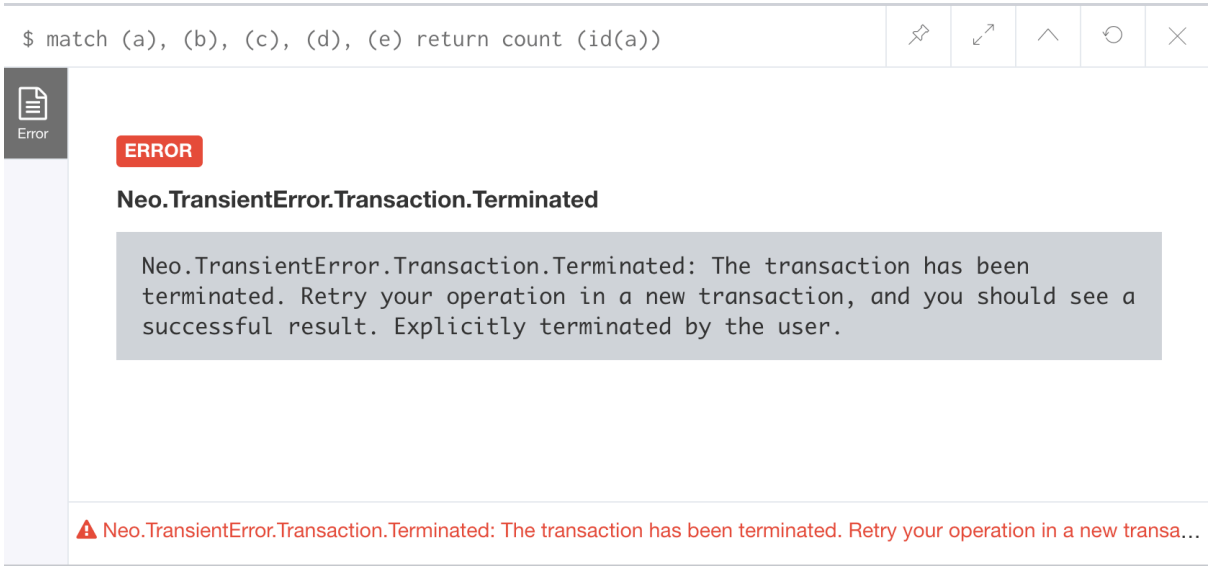
If the query executes and then **returns a lot of data**, there is no way to monitor it or kill the query. All that you can do is close your Neo4j Browser session and start a new one. If the server has many of these **rogue** queries running, it will slow down considerably so you should aim to limit 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. Here is a screenshot where we are monitoring a long-running query in another Neo4j Browser session:



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')`.

Here is what happens in the Neo4j Browser session where the long-running query was run:



NOTE	The `:queries` command is only available in the Enterprise Edition of Neo4j.

Exercise 13: Analyzing and monitoring queries

In the query edit pane of Neo4j Browser, execute the browser command: `:play intro-neo4j-exercises` and follow the instructions for Exercise 13.

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.

Constraints and node keys that enforce uniqueness are related to indexes which you will learn about later in this module.

NOTE	Existence constraints and node keys are only available in Enterprise Edition of Neo4j.

Ensuring that a property value for a node is unique

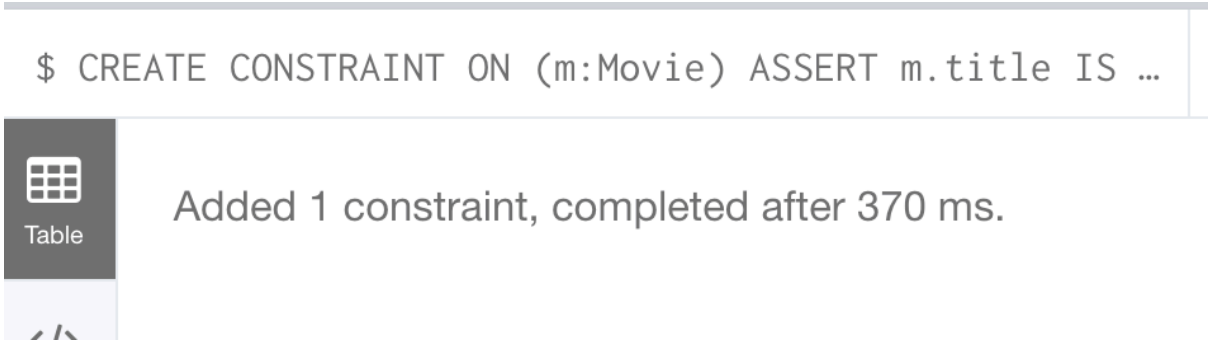
You add a uniqueness constraint to the graph by creating a constraint that asserts that a particular node property is unique in the graph for a particular type of node.

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.

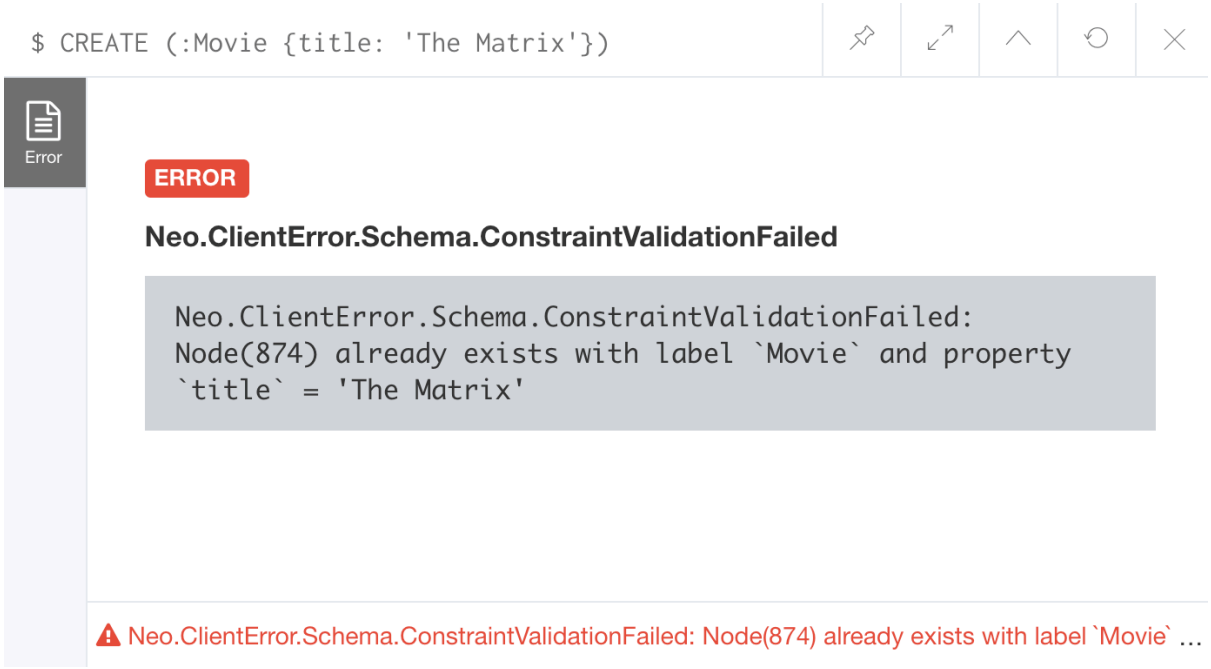
Here is the result of running this Cypher statement on the *Movie* graph:



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:



In addition, if you attempt to modify the value of a property where the uniqueness assertion fails, the property will not be updated.

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.

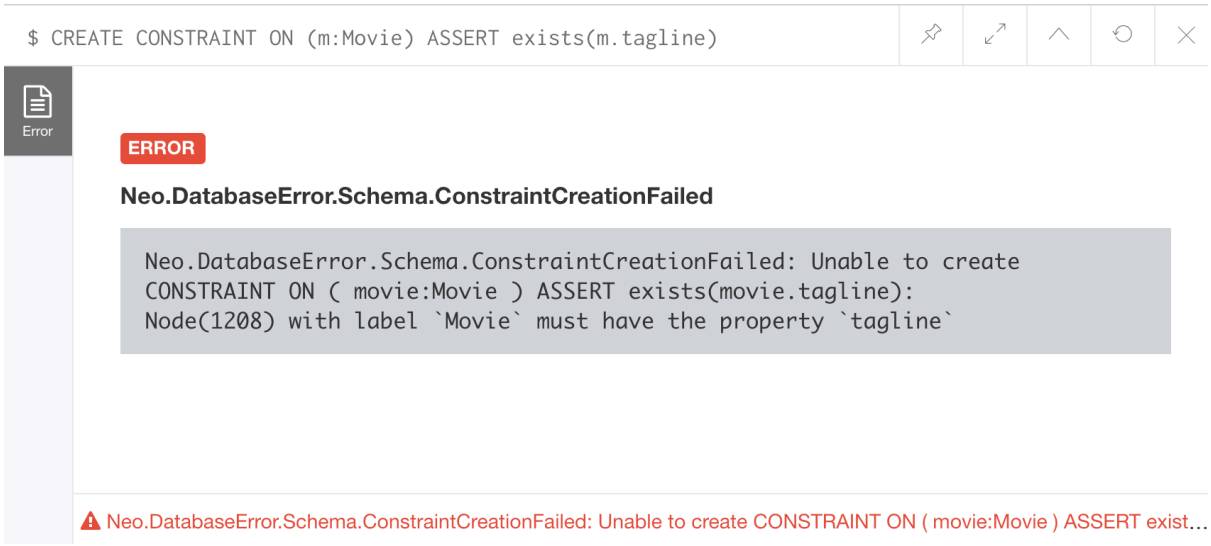
Recall that in the *Movie* graph, the movie, *Something's Gotta Give* has no *tagline* property:

```
$ MATCH (m:Movie) WHERE m.title STARTS WITH 'Something' RETURN m
```

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:



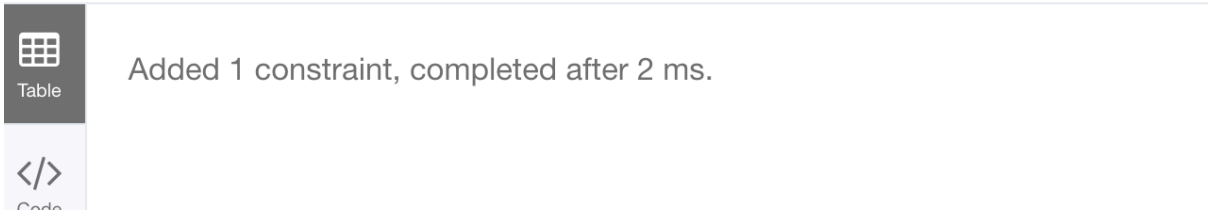
The constraint cannot be added to the graph because a node has been detected that violates the constraint.

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. With the result:

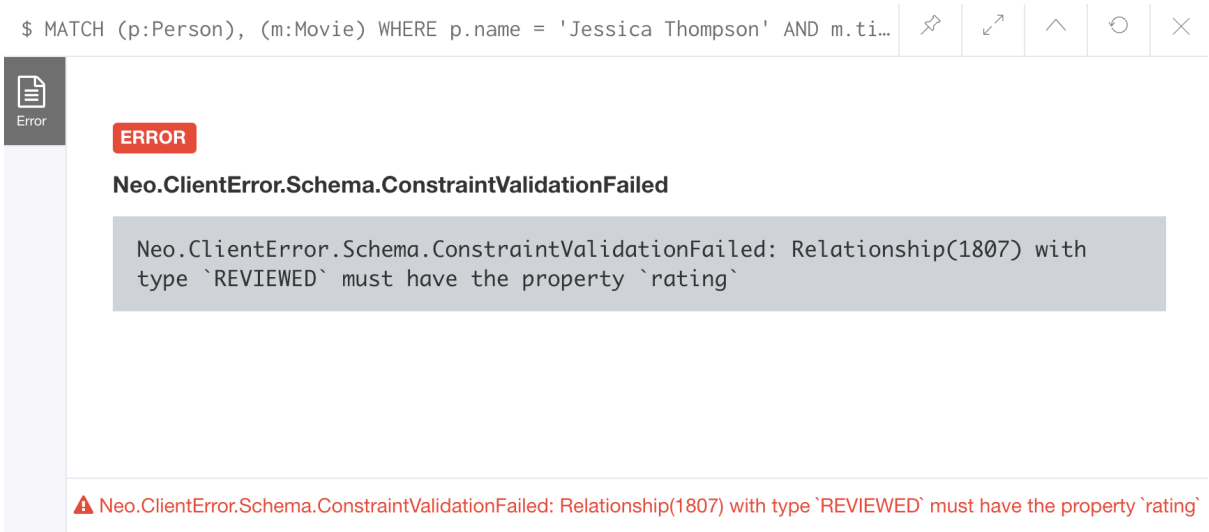
```
$ CREATE CONSTRAINT ON ()-[rel:REVIEWED]-() ASSERT exists(rel.rating)
```



So after creating this constraint, if we attempt to create a *:REVIEWED* relationship without setting the *rating* property:

```
MATCH (p:Person), (m:Movie)
WHERE p.name = 'Jessica Thompson' AND
      m.title = 'The Matrix'
MERGE (p)-[:REVIEWED {summary: 'Great movie!'}]->(m)
```

We see this error:



You will also see this error if you attempt to remove a property from a node or relationship where the existence constraint has been created in the graph.

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()
```

And here is what is returned from the graph:

```
$ CALL db.constraints
```

Table

description
"CONSTRAINT ON (movie:Movie) ASSERT movie.title IS UNIQUE"
"CONSTRAINT ON ()-[reviewed:REVIEWED]-() ASSERT exists(reviewed.rating)"

Text

NOTE

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)
```

With the result:

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

Table

Removed 1 constraint, completed after 1 ms.

Creating node keys

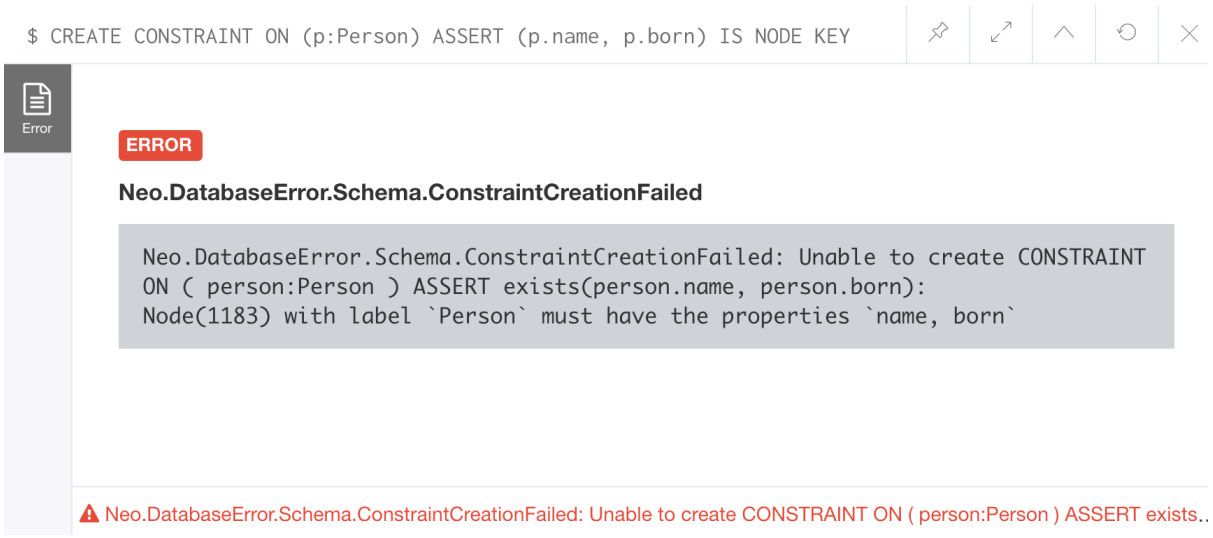
A node key is used to define the uniqueness constraint for multiple properties of a node of a certain type. A node key is also used as a composite index in the graph.

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:



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
```

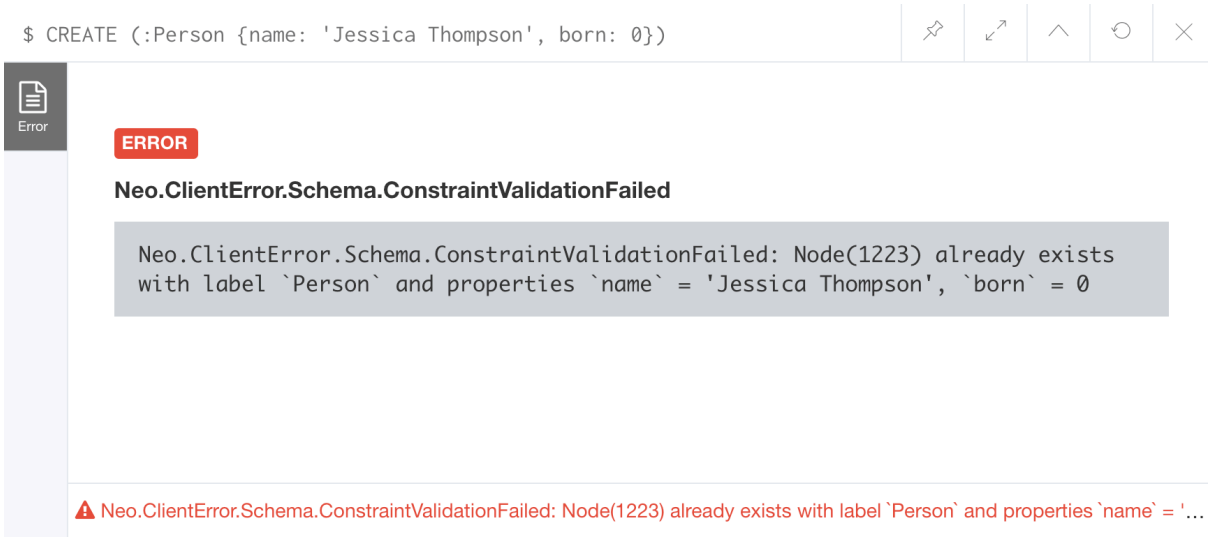
Then the creation of the node key succeeds:

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.

For example, executing this Cypher statement will fail:

```
CREATE (:Person {name: 'Jessica Thompson', born: 0})
```

Here is the result:



Exercise 14: Managing constraints and node keys

In the query edit pane of Neo4j Browser, execute the browser command: `:play intro-neo4j-exercises` and follow the instructions for Exercise 14.

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.

In this module, we introduce the basics of Neo4j indexes, but you should consult the Neo4j *Operations Manual* for more details about creating and maintaining indexes.

NOTE	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.

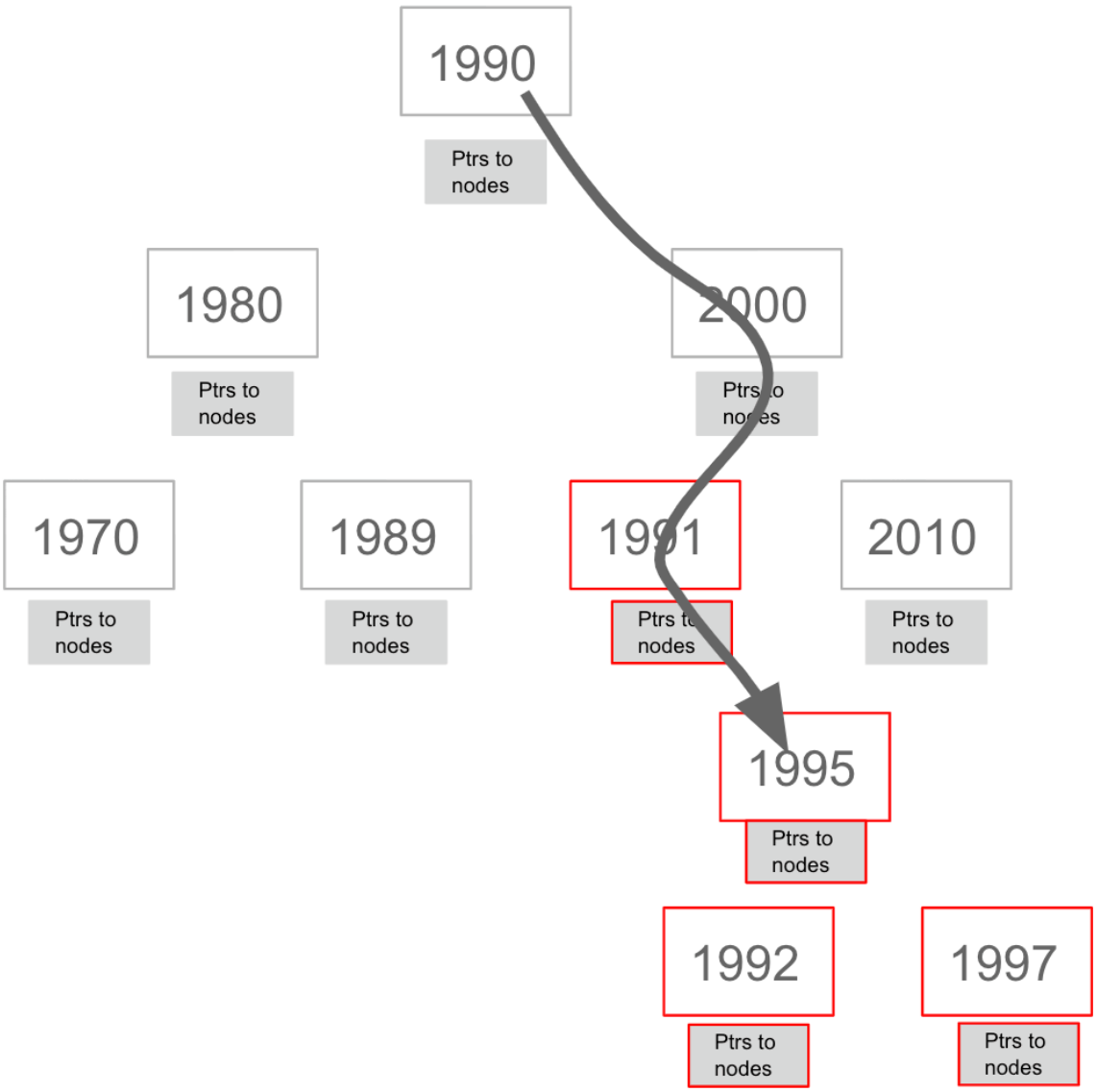
Indexes for range searches

When you add an index for a property of a node, it can greatly reduce the number of nodes the graph engine needs to visit in order to satisfy a query.

In this query we are testing the value of the *released* property of a *Movie* node using ranges:

```
MATCH (m:Movie)
WHERE 1990 < m.released < 2000
SET m.videoFormat = 'DVD'
```

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



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)
```

With the result:

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

Added 1 index, completed after 4 ms.

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.

Suppose we added the property, *videoFormat* to every *Movie* node and set its value, based upon the released date of the movie as follows:

```
MATCH (m:Movie)
WHERE m.released >= 2000
SET m.videoFormat = 'DVD';
MATCH (m:Movie)
WHERE m.released < 2000
SET m.videoFormat = 'VHS'
```

With the result:

NOTE

Notice that in the above Cypher statements we use the semi-colon `;` to separate Cypher statements. 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)
```

With the result:

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

Table

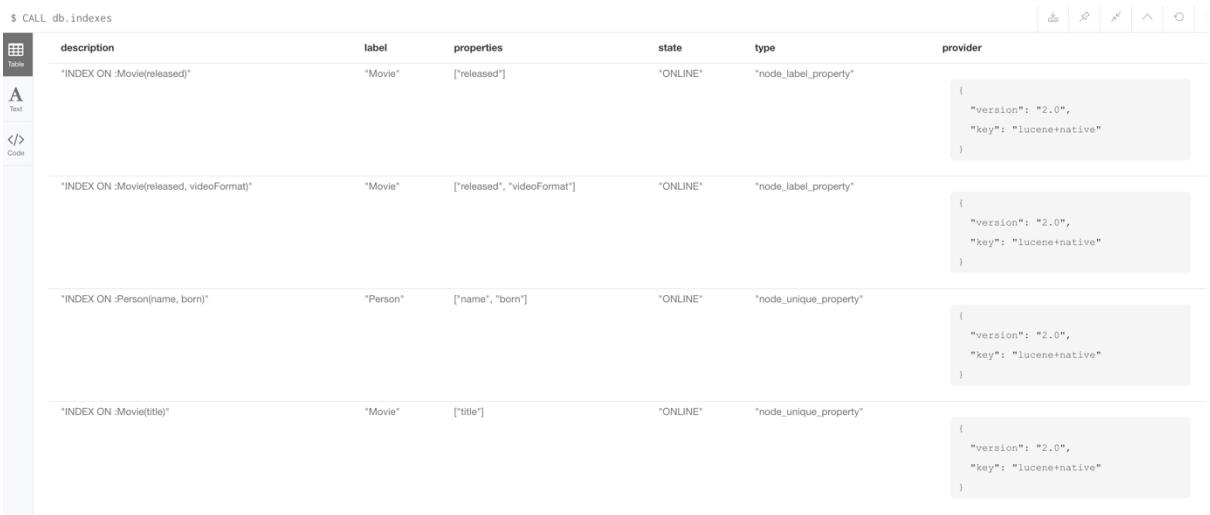
Added 1 index, completed after 2 ms.

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()
```

With the result:



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

Dropping indexes

You can drop an existing index that you created with `CREATE INDEX`.

Here is an example of dropping the composite index that we just created:

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

Here is the result:

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

Removed 1 index, completed after 8 ms.

Exercise 15: Managing indexes

In the query edit pane of Neo4j Browser, execute the browser command: `:play intro-neo4j-exercises` and follow the instructions for Exercise 15.

Going from relational to graph with Neo4j

In this video, you will learn how developers use Neo4j for implementing all or part of their relational models.

Importing data

In many applications, it is the case that the data that you want to populate your graph with comes from data that was written to *.csv* files or files of other types. There are many nuances and best practices for loading data into a graph from files. 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 http://neo4j.com/developer/neo4j-etl/, 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`

Cypher provides an elegant built-in way to import tabular CSV data into graph structures.

The `LOAD CSV` clause parses a local 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`:

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

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:

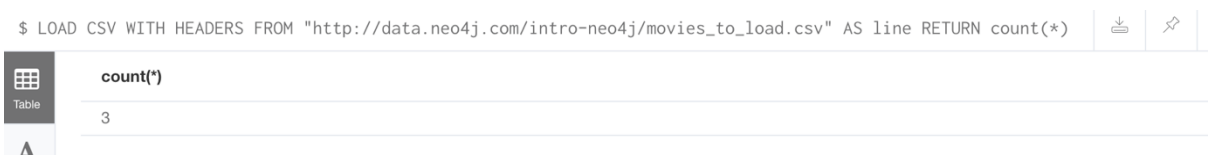
```
id,title,country,year,summary
1,Wall Street,USA,1987, Every dream has a price.
2,The American President,USA,1995, Why can't the most powerful man in the world have the one thing he wants most?
3,The Shawshank Redemption,USA,1994, Fear can hold you prisoner. Hope can set you free.
```

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(*)
```

Here is the count result for this particular file:



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
```

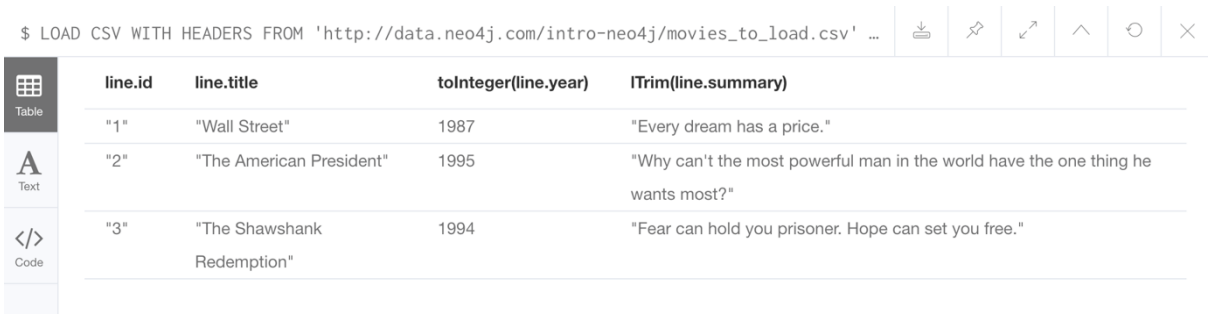
Here is the result of running the Cypher statement to visually inspect the data:

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)
```

Here we see how the data will be formatted before it is loaded:



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.

Here is the result of loading the **movies_to_load. csv** data into the graph:

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
```

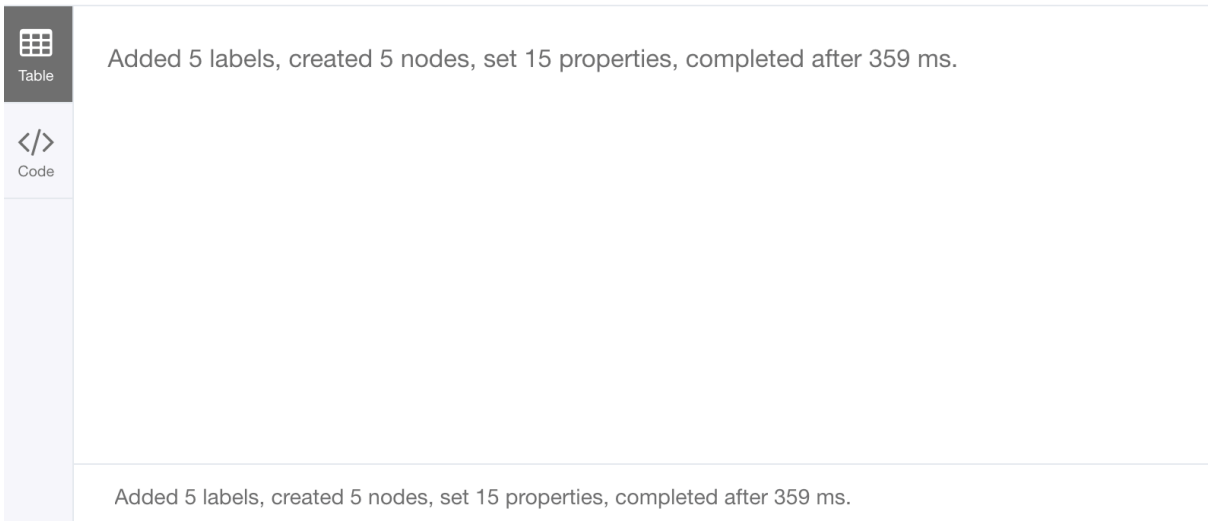
In case you already have people in your database, you will want to avoid creating duplicates. That's why instead of just creating them, we use `MERGE` to ensure unique entries after the import. We use the `ON CREATE` clause to set the values for *name* and *born*.

```
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 as 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.

Here is the result of loading the **persons_to_load. csv** data into the graph:

```
$ LOAD CSV WITH HEADERS FROM 'http://data.neo4j.com/intro-neo4j/persons_to_load.csv' AS li…
```



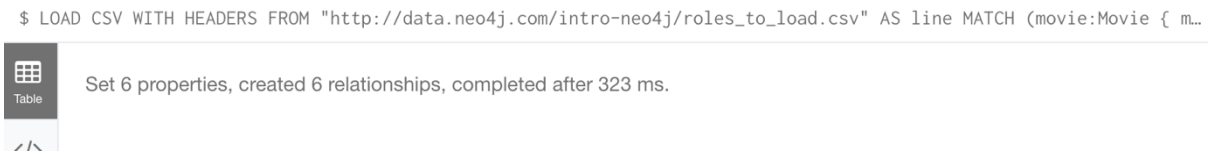
The **roles_to_load.csv** file (sample below) holds the data that will populate the relationships between the nodes.

```
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)
```

Here is the result of loading the **roles_to_load. csv** data into the graph:



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:

movie_actor_roles_to_load.csv:

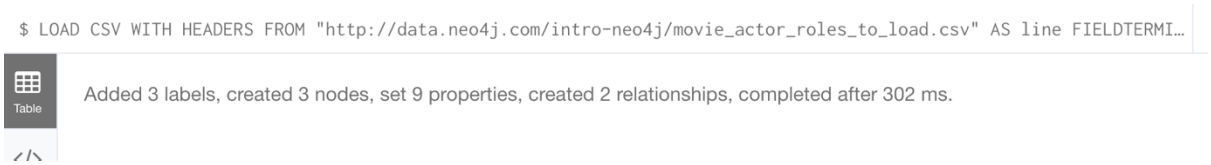
```
title;released;summary;actor;birthyear;characters

Back to the Future;1985;17 year old Marty McFly got home early last night. 30 years early.;Michael J. Fox;1961;Marty McFly

Back to the Future;1985;17 year old Marty McFly got home early last night. 30 years early.;Christopher Lloyd;1938;Dr. Emmet Brown
```

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.

Here is the result of loading the **movie_actor_roles_to_load. csv** data into the graph:



For large denormalized files, it may still make sense to create nodes and relationships separately in multiple passes. That would depend on the complexity of the operations and the experienced performance.

Importing a large dataset

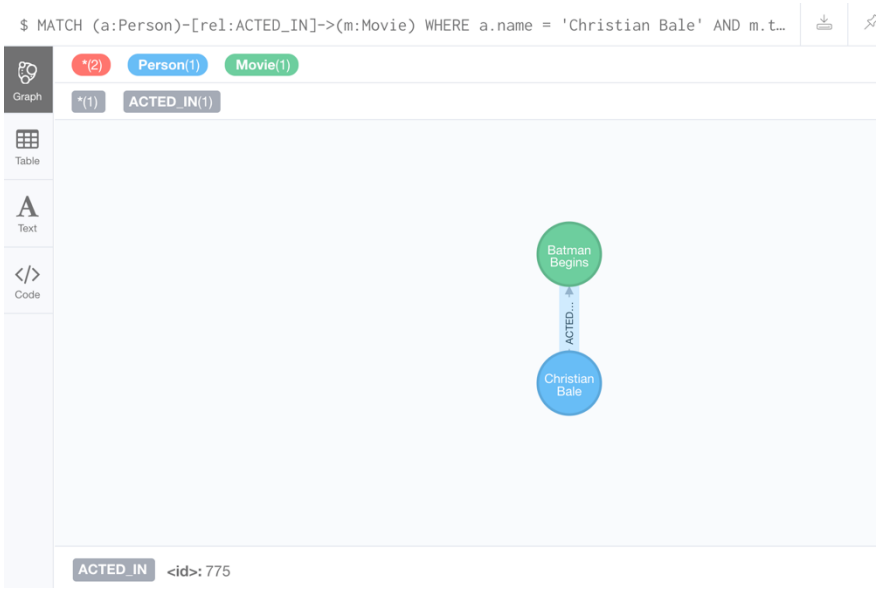
If you import a larger amount of data (more than 10,000 rows), it is recommended to prefix your `LOAD CSV` clause with a `PERIODIC COMMIT` hint. This allows the database to regularly commit the import transactions to avoid memory churn for large transaction-states.

Exercise 16: Importing data

In the query edit pane of Neo4j Browser, execute the browser command: `:play intro-neo4j-exercises` and follow the instructions for Exercise 16.

Accessing Neo4j resources

There are many ways that you can learn more about Neo4j. A good starting point for learning about the resources available to you is the **Neo4j Learning Resources** page at https://neo4j.com/developer/resources/.



Exercise 9: Creating Relationships

In the query edit pane of Neo4j Browser, execute the browser command: `:play intro-neo4j-exercises` and follow the instructions for Exercise 9.

Deleting nodes and relationships

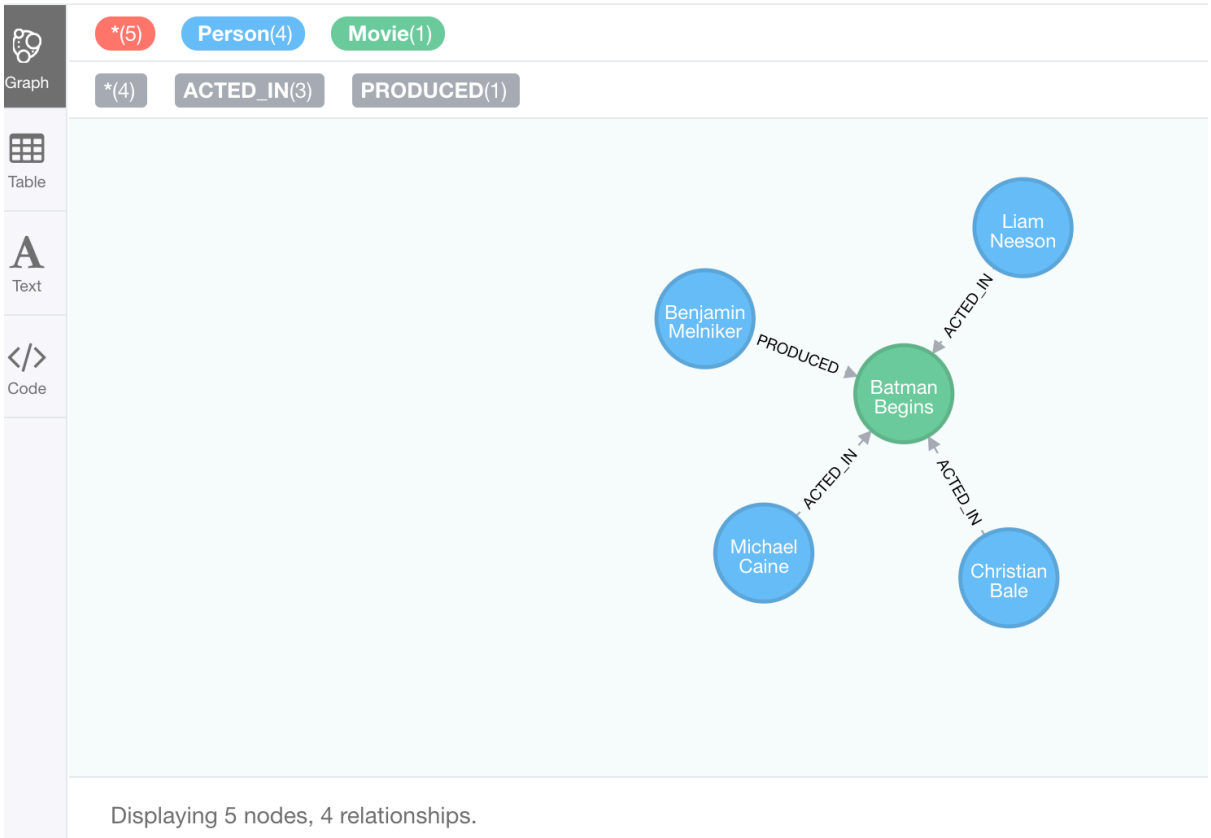
If a node has no relationships to any other nodes, you can simply delete it from the graph using the `DELETE` clause. Relationships are also deleted using the `DELETE` clause.

NOTE	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

Here are the existing nodes and relationships for the *Batman Begins* movie:

```
$ MATCH (a:Person)-[rel]->(m:Movie) WHERE m.title = 'Batman Begins' RETURN a, rel, m
```

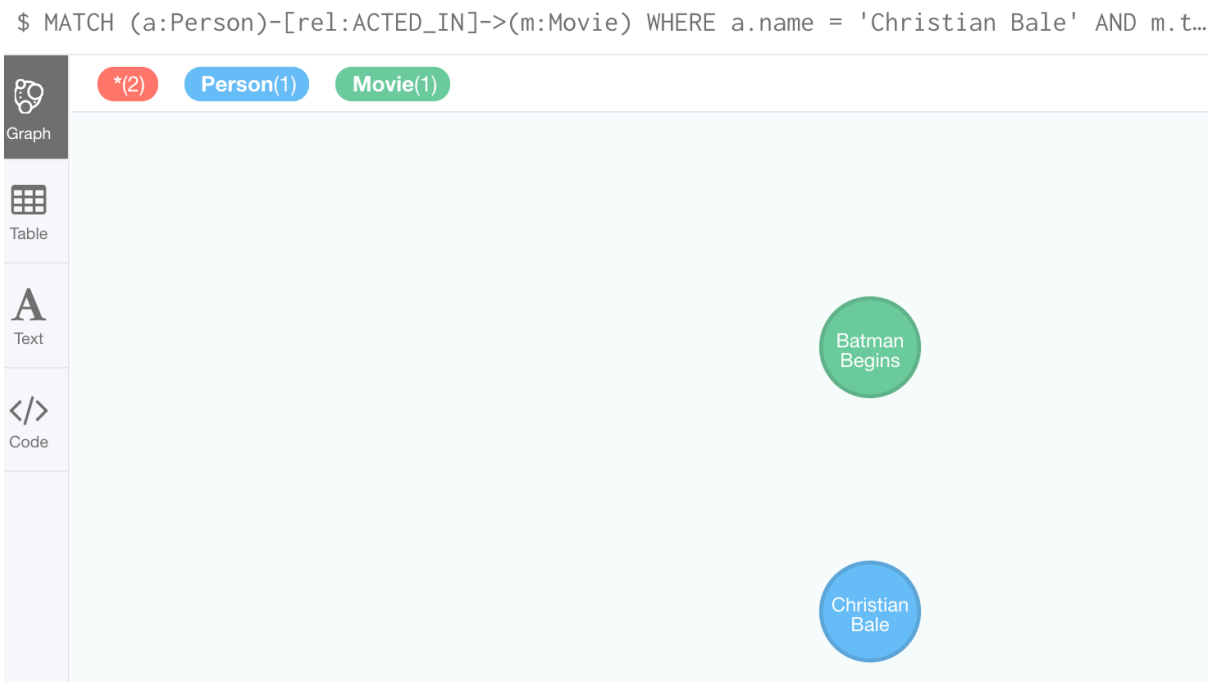


You can delete a relationship between nodes by first finding it in the graph and then deleting it.

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
```

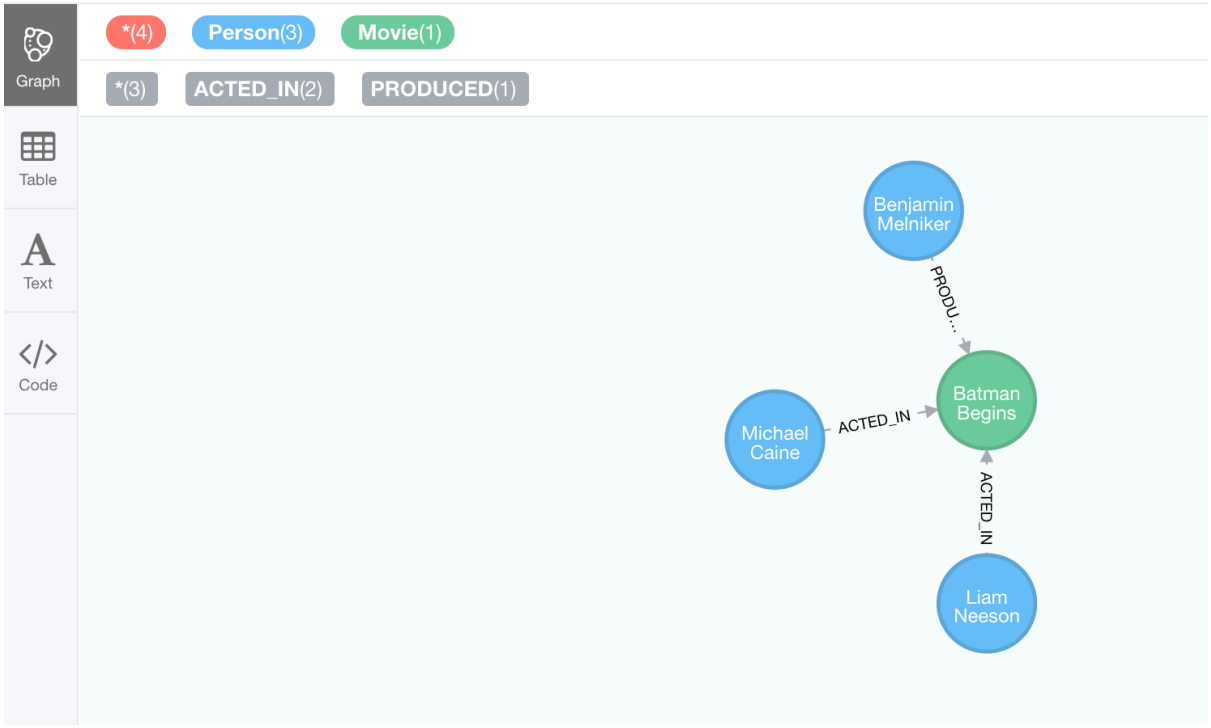
Here is the result of running this Cypher statement:



Notice that there no longer exists the relationship between *Christian Bale* and the movie *Batman Begins*.

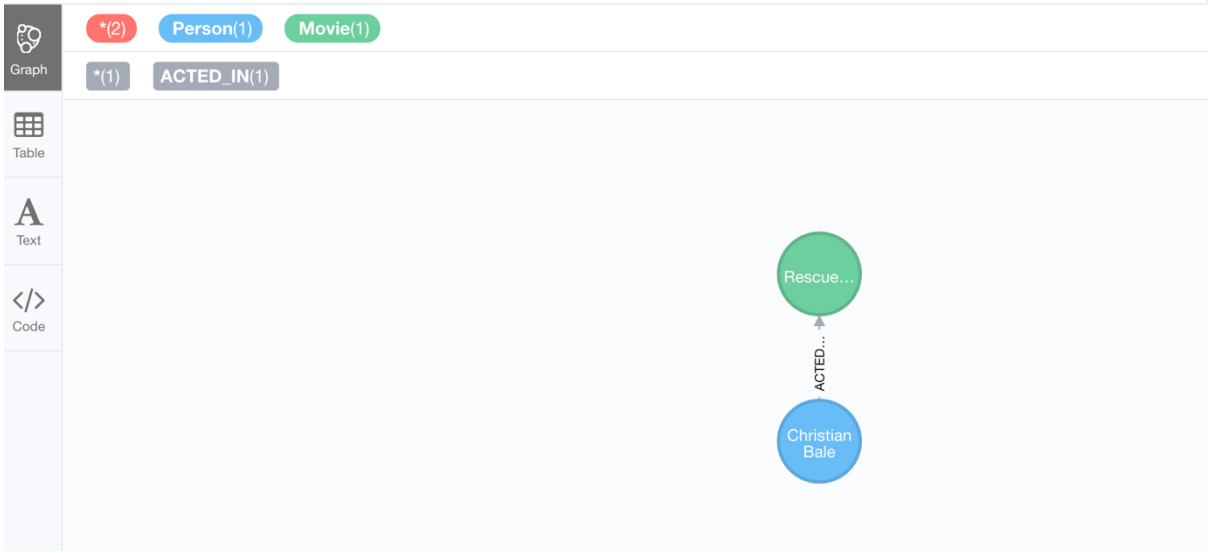
We can now query the nodes related to *Batman Begins* to see that this movie now only has two actors and one producer connected to it:

```
$ MATCH (a:Person)-[rel]->(m:Movie) WHERE m.title = 'Batman Begins' RETURN a, rel, m
```



Even though we have deleted the relationship between actor, *Christian Bale* and the movie *Batman Begins*, we note that this actor is connected to another movie in the graph, so we should not delete this *Christian Bale* node.

```
$ MATCH (a:Person)-[rel:ACTED_IN]->(m:Movie) WHERE a.name = 'Christian Bale'  RETURN a, rel, 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]->(:Movie)
WHERE p.name = 'Benjamin Melniker'
```

```
DELETE rel, p
```

Here is the result of running this Cypher statement:

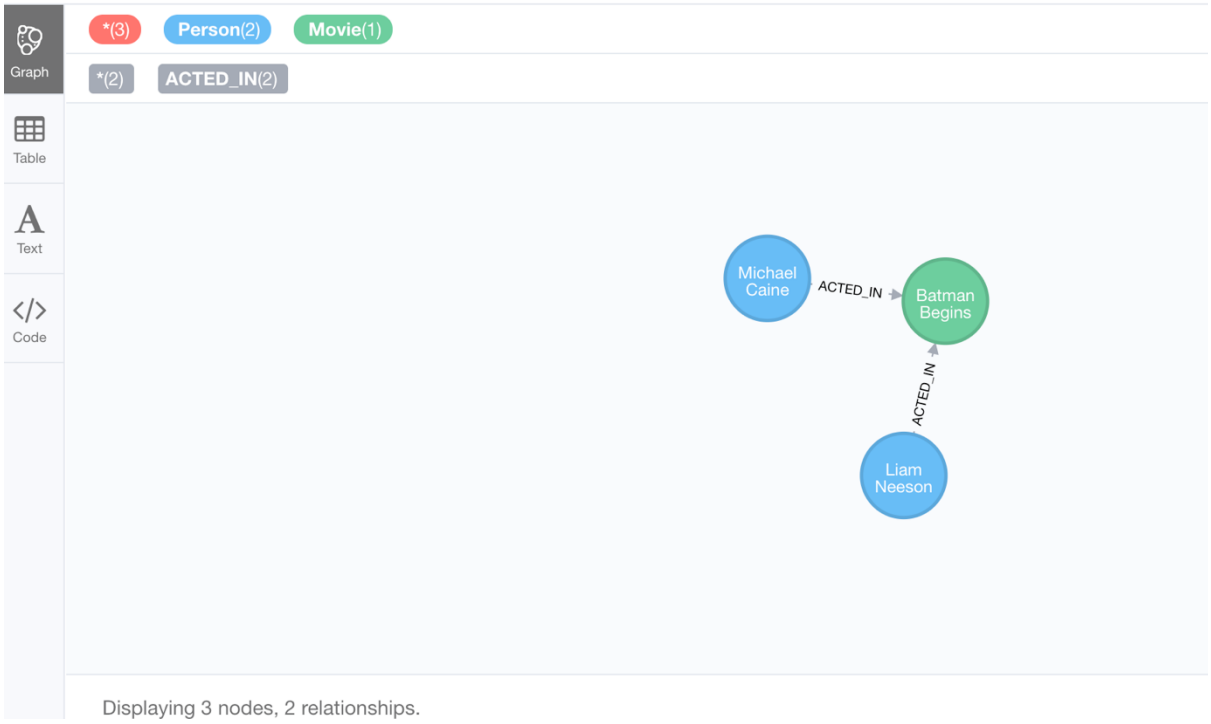
```
$ MATCH (p:Person)-[rel:PRODUCED]->(:Movie) WHERE p.name = 'Benjamin Melniker' DELETE rel, p
```

Table

Deleted 1 node, deleted 1 relationship, completed after 3 ms.

And here we see that we now have only two connections to the *Batman Begins* movie:

```
$ MATCH (a:Person)-[rel]->(m:Movie) WHERE m.title = 'Batman Begins' RETURN a, rel, m
```



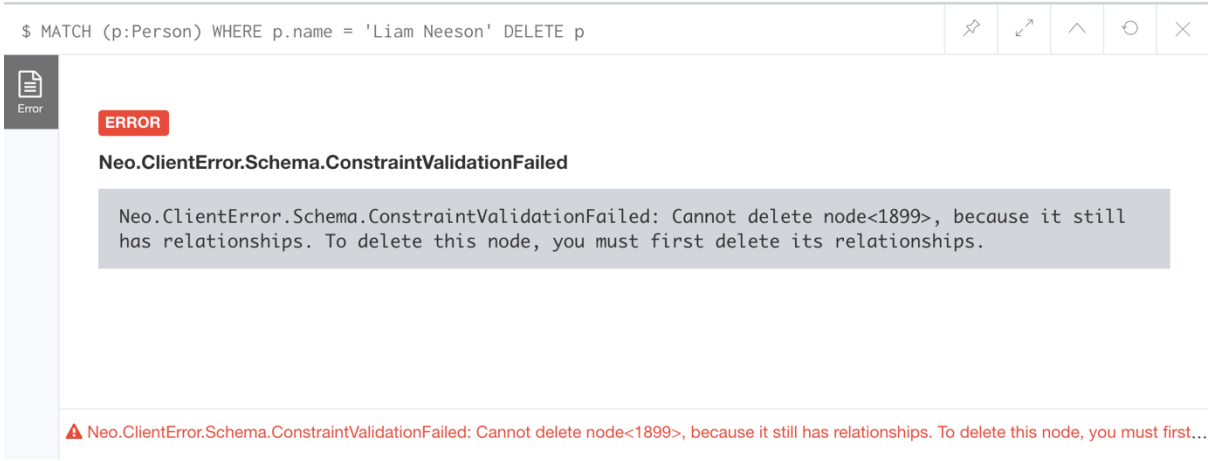
Deleting nodes and relationships

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 were were to attempt to delete the *Liam Neeson* node without first deleting its relationships:

```
MATCH (p:Person)
WHERE p.name = 'Liam Neeson'
DELETE p
```

We would see this error:

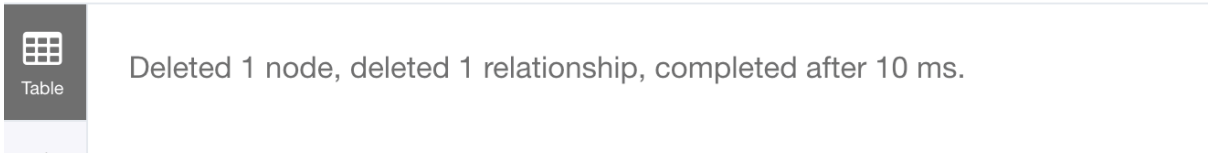


Here we delete the *Liam Neeson* node and its relationships to any other nodes:

```
MATCH (p:Person)
WHERE p.name = 'Liam Neeson'
DETACH DELETE  p
```

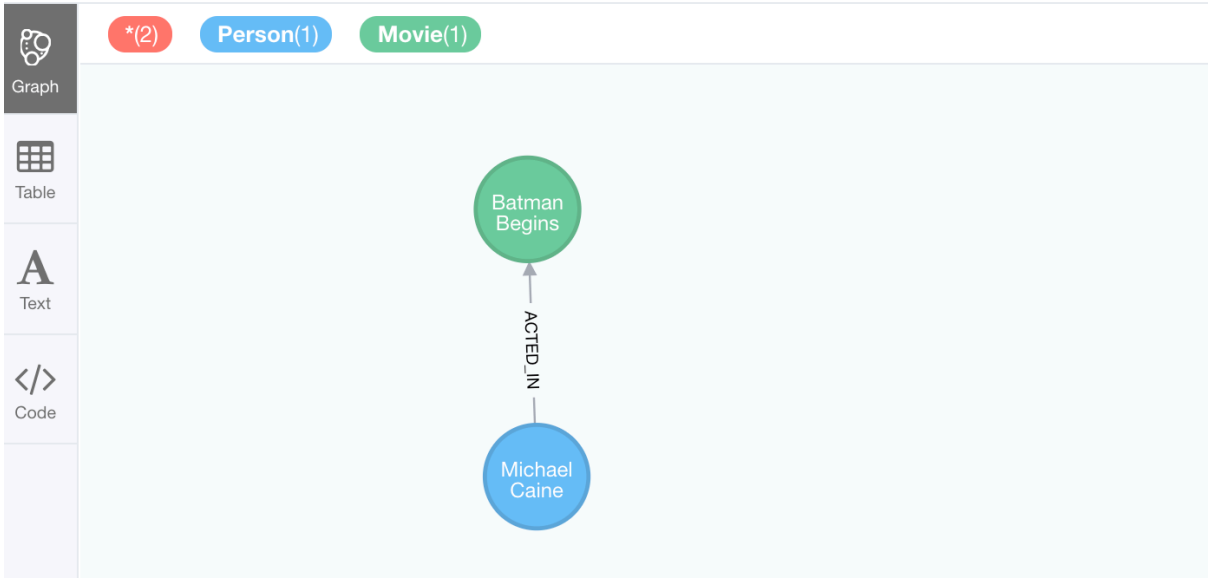
Here is the result of running this Cypher statement:

```
$ MATCH (p:Person) WHERE p.name = 'Liam Neeson' DETACH DELETE p
```



And here is what the *Batman Begins* node and its relationships now look like. There is only one actor, *Michael Caine* connected to the movie.

```
$ MATCH (a:Person)--(m:Movie) WHERE m.title = 'Batman Begins' RETURN a, m
```



Exercise 10: Deleting Nodes and Relationships

In the query edit pane of Neo4j Browser, execute the browser command: `:play intro-neo4j-exercises` and follow the instructions for Exercise 10.

Merging data in the graph

Thus far, you have learned how to create nodes, labels, properties, and relationships in the graph. You can use `MERGE` to either create new nodes and relationships or to make structural changes to existing nodes and relationships.

For exmaple, how the graph engine behaves when a duplicate element is created depends on the type of element:

If you use `CREATE`:	**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. **Note:** If you specify a set of properties to be created using `=` rather than `+=`, it could remove existi they are not included in the set.
Relationship	If the relationship exists, a duplicate relationship is created.

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

The `MERGE` clause is used to find elements in the graph. But 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.

When you specify *nodeProperties* for `MERGE`, you should only use properties that satisfy some sort of uniqueness constraint. You will learn about uniqueness constraints in the next module.

Here is what we currently have in the graph for the *Person*, *Michael Caine*. This node has values for *name* and *born*. Notice also that the label for the node is *Person*.

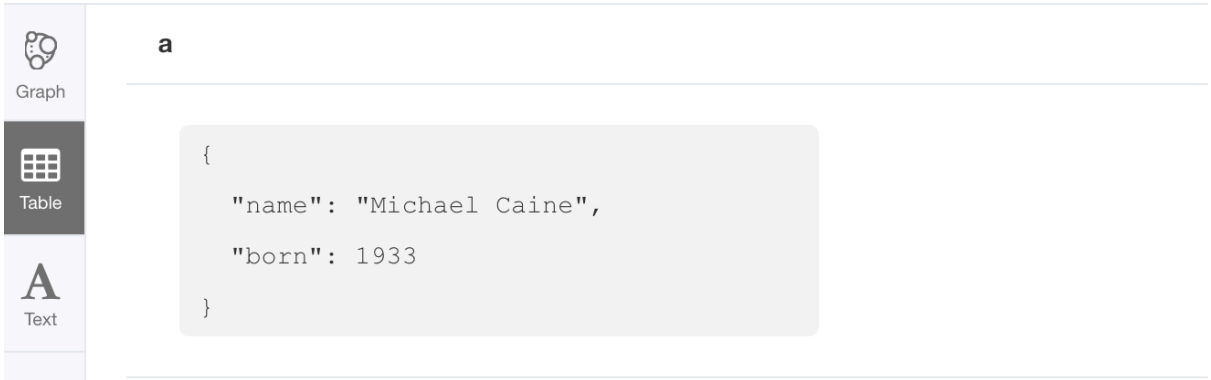
```
$ MATCH (a:Person {name: 'Michael Caine', born: 1933}) RETURN a
```

Here we use `MERGE` to find a node with the *Actor* label with the key property *name* of *Michael Caine*, and we set the *born* property to *1933*. Our data model has never used the label, *Actor* so this is a new entity type in our graph.

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

Here is the result of running this Cypher example. We do not find a node with the label *Actor* so the graph engine creates one.

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

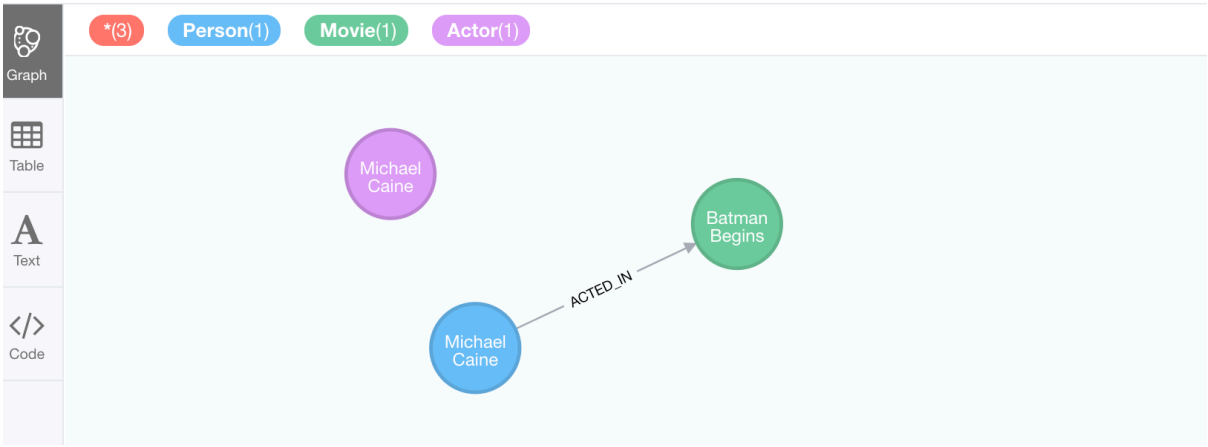


NOTE	When you specify the node to merge, you should only use properties that have a unique index. You will learn about uniqueness later in this training.

If we were to repeat this `MERGE` clause, no additional *Actor* nodes would be created in the graph.

At this point, however, we have two *Michael Caine* nodes in the graph, one of type *Person*, and one of type *Actor*:

```
$ MATCH (a {name: 'Michael Caine'}), (m:Movie) WHERE m.title = 'Batman Begins' RETURN a, m
```



NOTE	Be mindful that node labels and the properties for a node are significant when merging nodes.

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

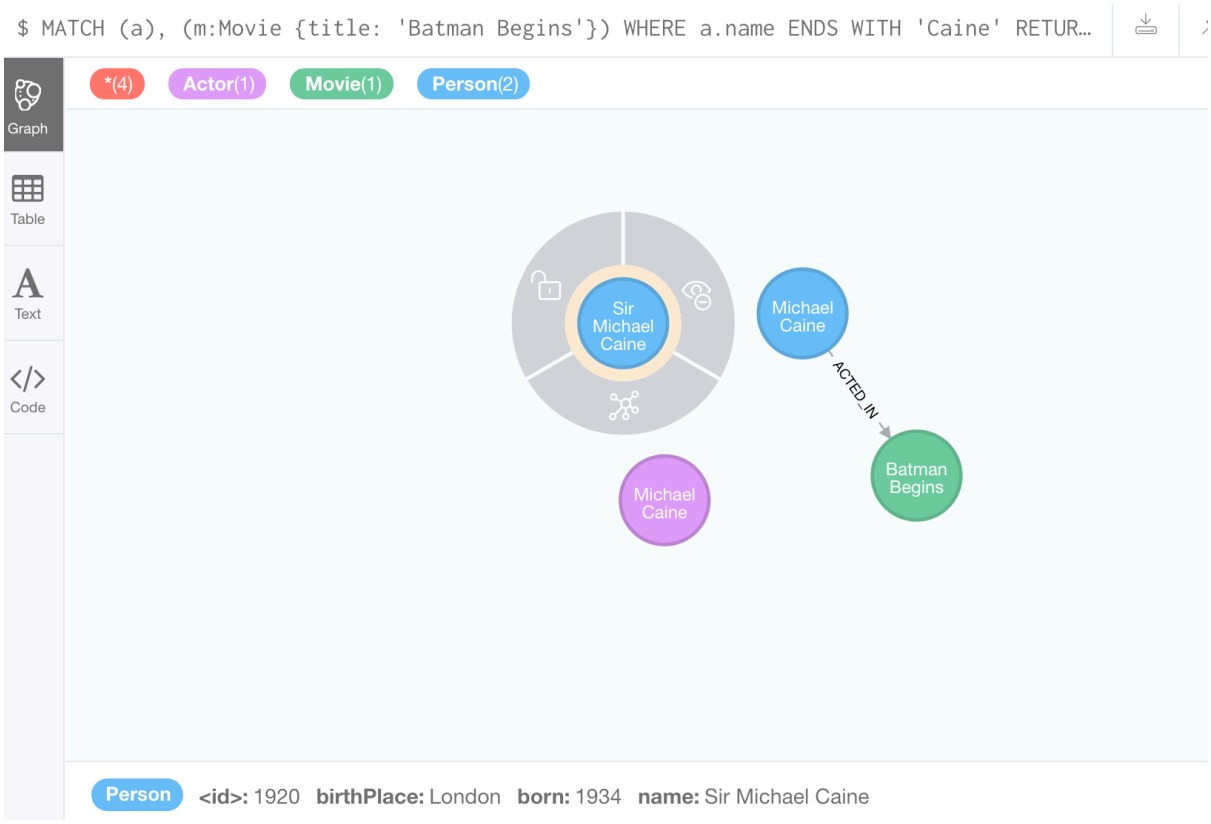
You can use the `MERGE` clause, along with `ON CREATE` to assign specific values to a node being created as a result of an attempt to merge.

Here is an example where create a new node, specifying property values for the new node:

```
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.

Here is the resulting nodes that have anything to do with *Michael Caine*. The most recently created node has the *name* value of *Sir Michael Caine*.

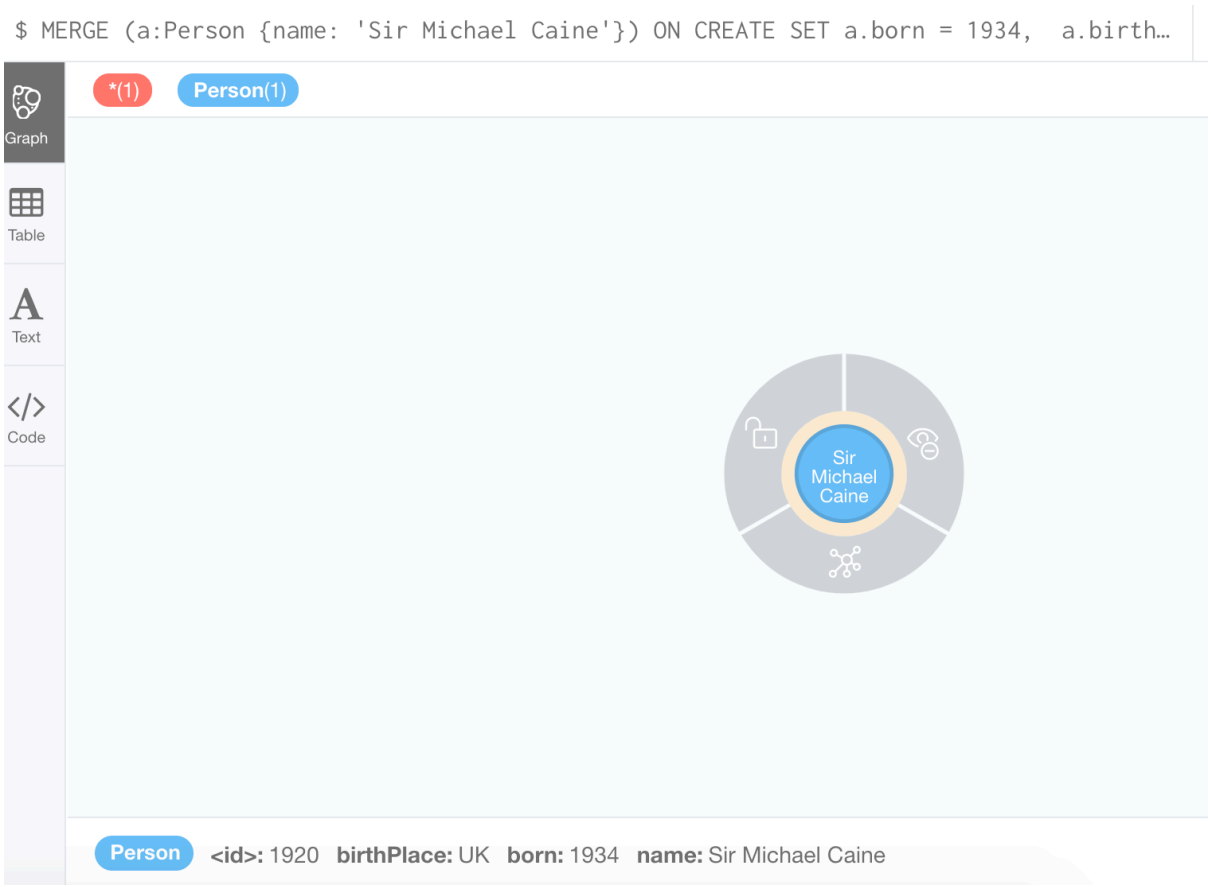


You can also specify an `ON MATCH` clause during merge processing. If the exact node is found, you can update its properties or labels. Here is an example:

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

```
ON CREATE SET a.born = 1934,
              a.birthPlace = 'UK'
ON MATCH SET a.birthPlace = 'UK'
RETURN a
```

And here we see that the found node (with the *<id>* of *1920*) was updated with the new value for *birthPlace*.



Using `MERGE` to create relationships

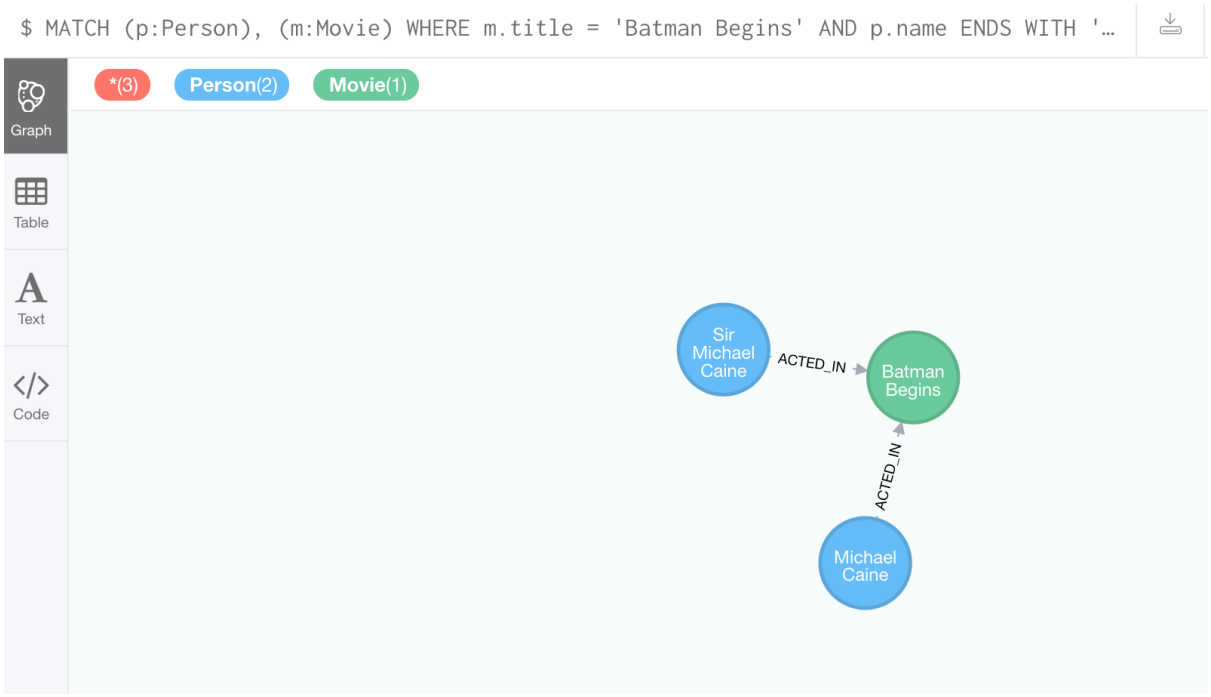
Using `MERGE` to create relationships is expensive and you should only do it when you need to ensure that a relationship is unique and you are not sure if it already exists.

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
```

Here is the result of executing this Cypher statement. It went through all the nodes and added the relationship to the nodes that didn't already have the relationship.



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.

For example:

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

Exercise 11: Merging Data in the graph

In the query edit pane of Neo4j Browser, execute the browser command: `:play intro-neo4j-exercises` and follow the instructions for Exercise 11.