Creating Nodes and Relationships

Table of Contents

About this module

You have learned how to query a graph using a number of Cypher clauses and keywords that start with the `MATCH` clause. You learned how to retrieve data based upon label values, property key values, and relationship types, and how to perform some useful intermediate processing during a query to control what data is returned.

At the end of this module, you should be able to write Cypher statements to:

- Create a node
  - Add and remove node labels
  - Add and remove node properties
  - Update properties
- Create a relationship
  - Add and remove properties for a relationship
- Delete a node
- Delete a relationship
- Merge data in a graph
  - Creating nodes
  - Creating 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.:


```
$ CREATE (m:Movie {title: 'Batman Begins'})  RETURN m
```
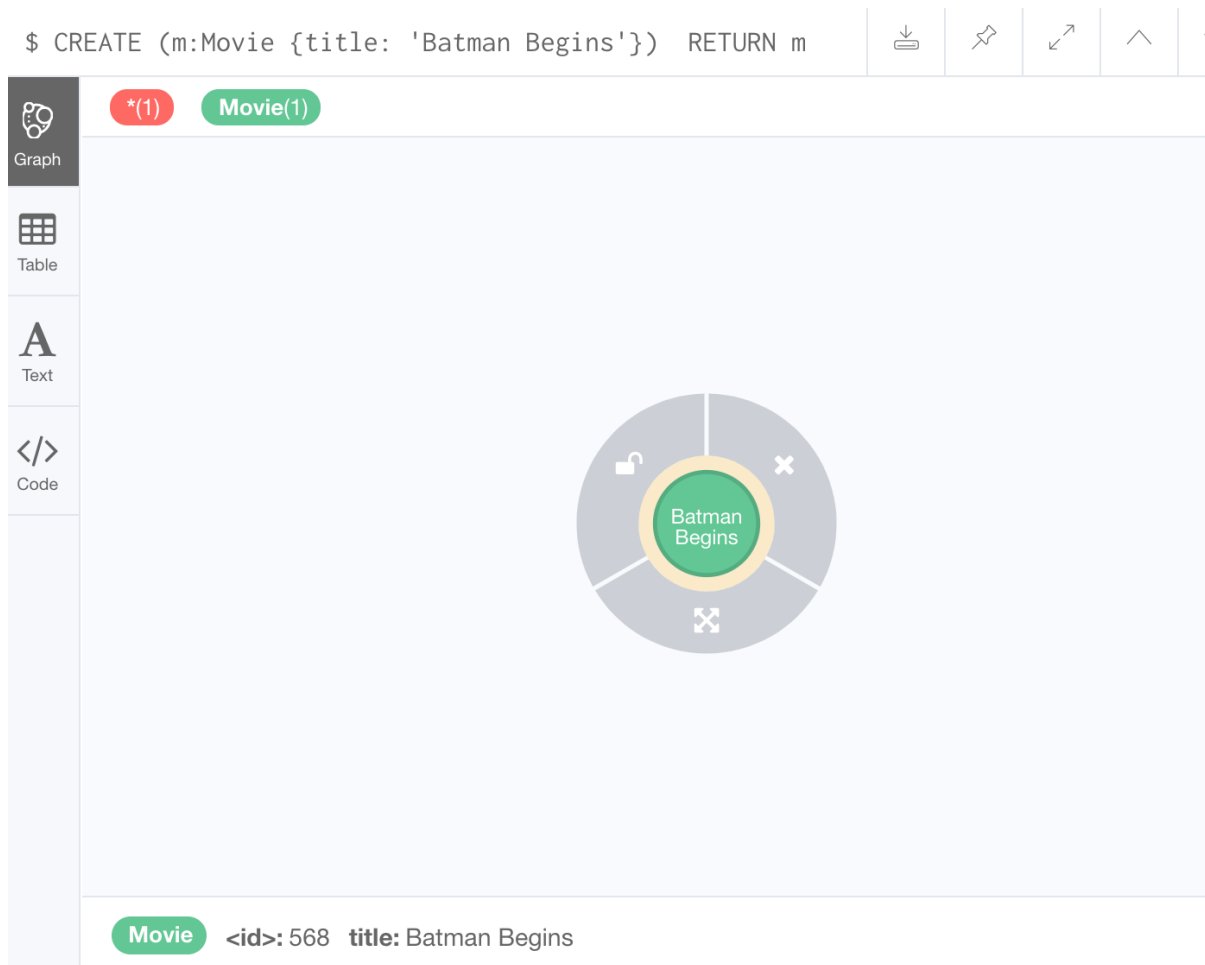
When the graph engine creates a node, it automatically assigns a read-only, unique ID to the node. Here we see that the *id* of the node is *568*. This is not a property of a node, but rather an internal value.

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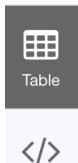
Added 4 labels, created 4 nodes, set 8 properties, completed after 1 ms.

| 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)

	labels(m)
Table	
	["Movie", "Action"]
A **Text**	
</> **Code**	

Added 1 label, started streaming 1 records after 8 ms and completed after 8 ms.

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:

$ MATCH (m:Movie:Action) WHERE m.title = 'Batman Begins' REMOVE m:Action RETURN labe…

	labels(m)
Table	["Movie"]
A Text	
>_ Code	

Removed 1 label, started streaming 1 records after 22 ms and completed after 22 ms.

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:

```
$ MATCH (m:Movie) WHERE m.title = 'Batman Begins' SET m.released = 2005, m.lengthInMinutes = 140 .
```

m

```
{
  "title": "Batman Begins",
  "lengthInMinutes": 140,
  "released": 2005
}
```

Set 2 properties, started streaming 1 records after 6 ms and completed after 6 ms.

Graph

Table

Text

Code

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 <u>all</u> 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…
```

	m
Graph	

Table	`{`
	`  "lengthInMinutes": 140,`
	`  "grossMillions": 206.5,`
	`  "title": "Batman Begins",`
	`  "videoFormat": "DVD",`
	`  "released": 2005`
	`}`

Text	
Code	

Set 5 properties, started streaming 1 records after 1 ms and completed after 1 ms.

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 <u>never</u> 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()
```

	propertyKey
Table	"title"
	"released"
A	"tagline"
Text	"name"
	"born"
</>	"roles"
Code	"summary"
	"rating"
	"id"
	"share_link"
	"favorite_count"
	"display_name"
	"lengthInMinutes"
	"videoFormat"
	"grossMillions"

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.

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

m

```
{
  "title": "Batman Begins",
  "lengthInMinutes": 140,
  "released": 2005
}
```

Set 2 properties, started streaming 1 records after 2 ms and completed after 2 ms.

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 <u>must</u> 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:
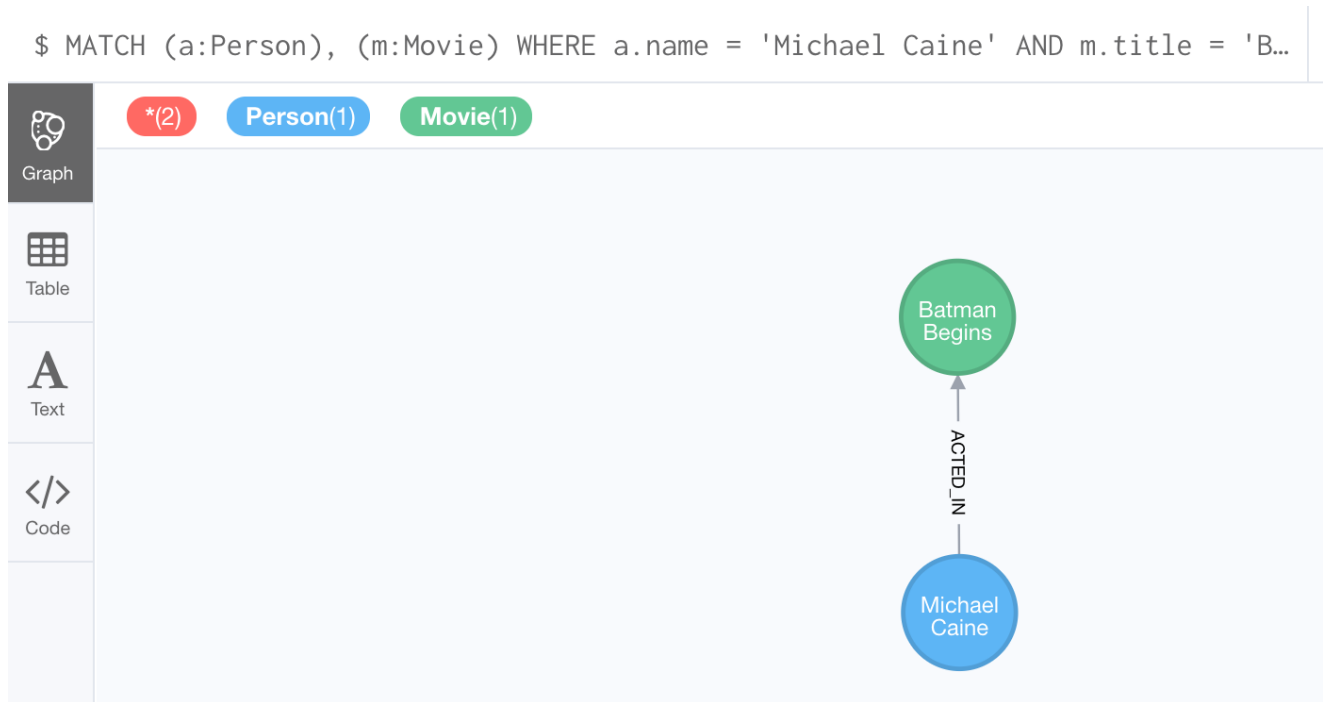
> **NOTE**
>
> Before you run these Cypher statements, you may see a warning in Neo4j Browser that you are creating a query that is a cartesian product that could potentially be a performance issue. You will see this warning if you have no unique constraint on the lookup keys. You will learn about uniqueness 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.
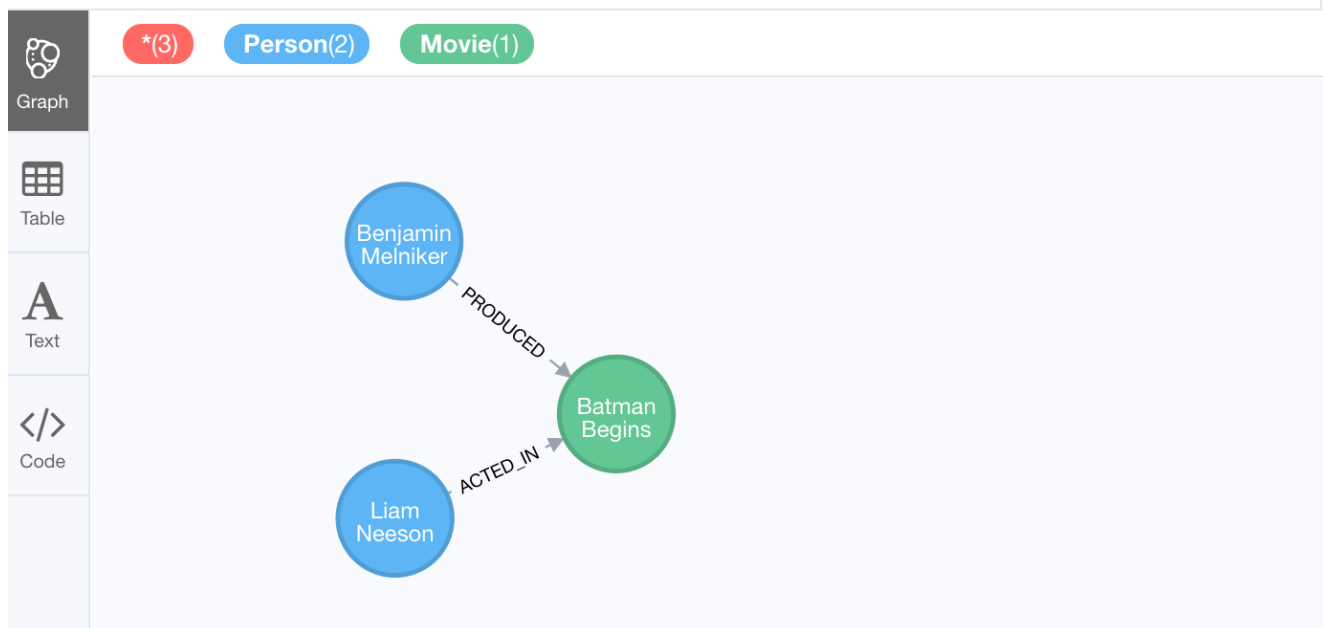
You can create multiple relationships at once by simply providing the pattern for the creation that includes the relationship types, their directions, and the nodes that you want to connect.

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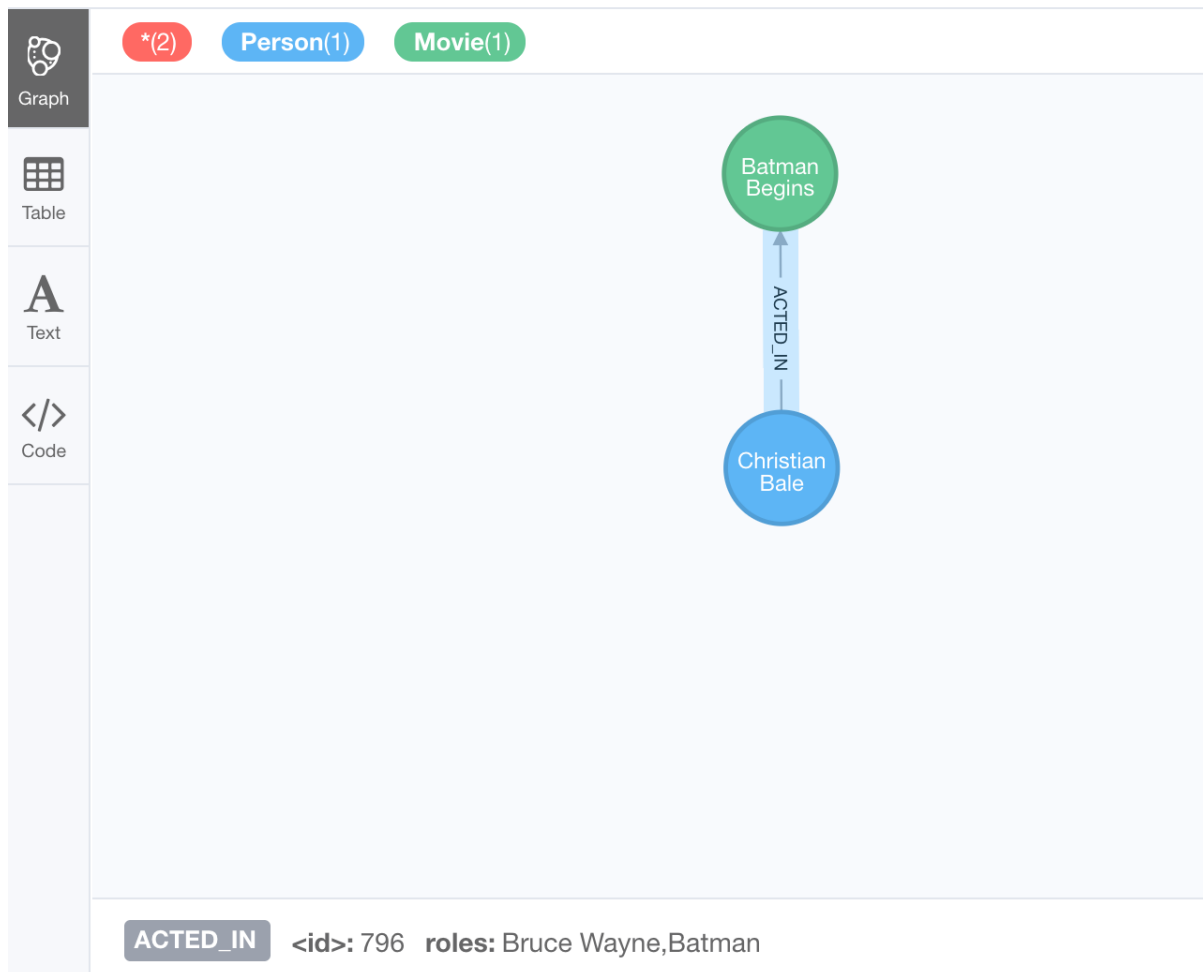
The *roles* property is a list so we add it as such. If the relationship had multiple properties, we could have added them as a comma separated list or as an object, like can do for node properties.

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`
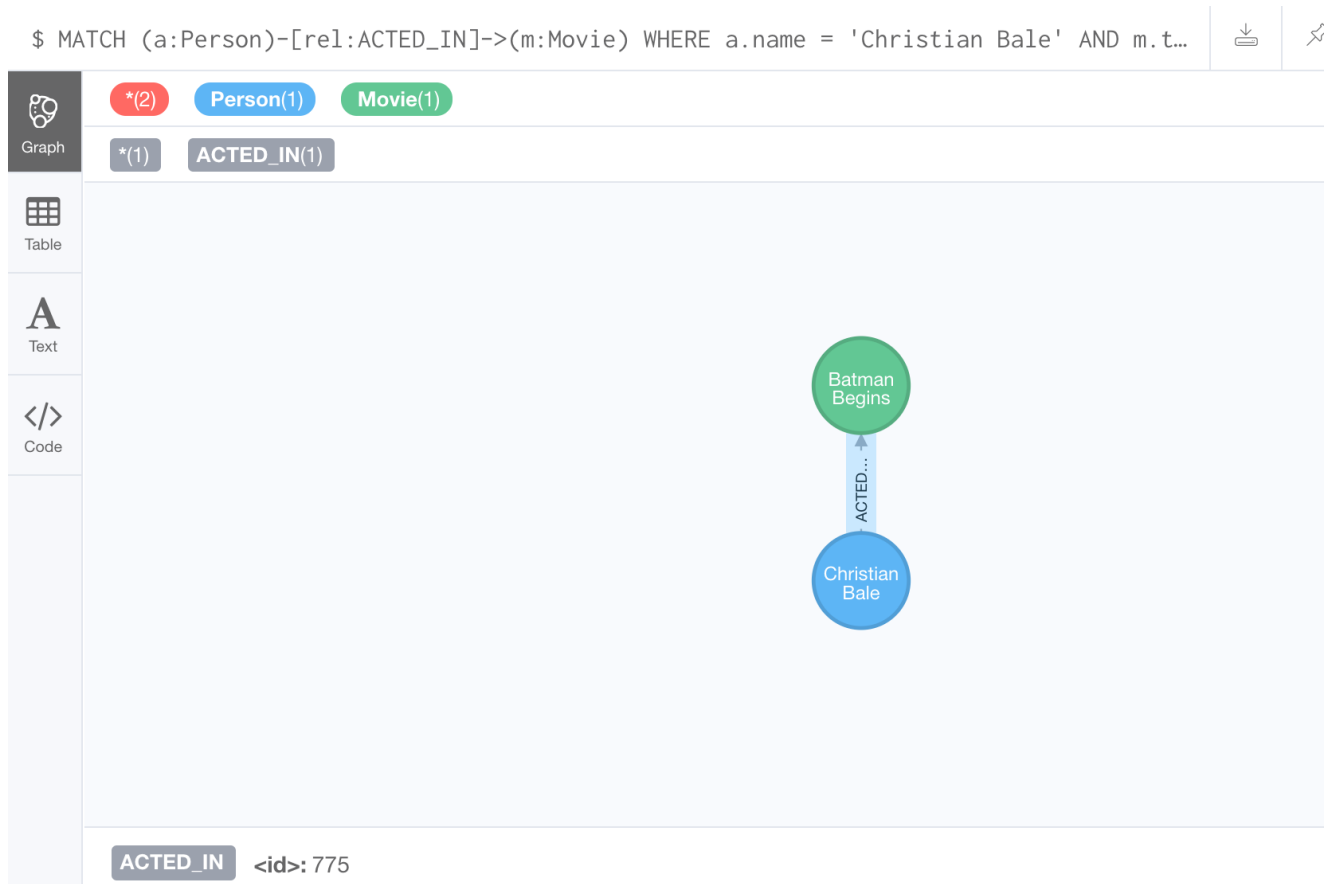
Exercise 9: Creating Relationships

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
```


Displaying 5 nodes, 4 relationships.
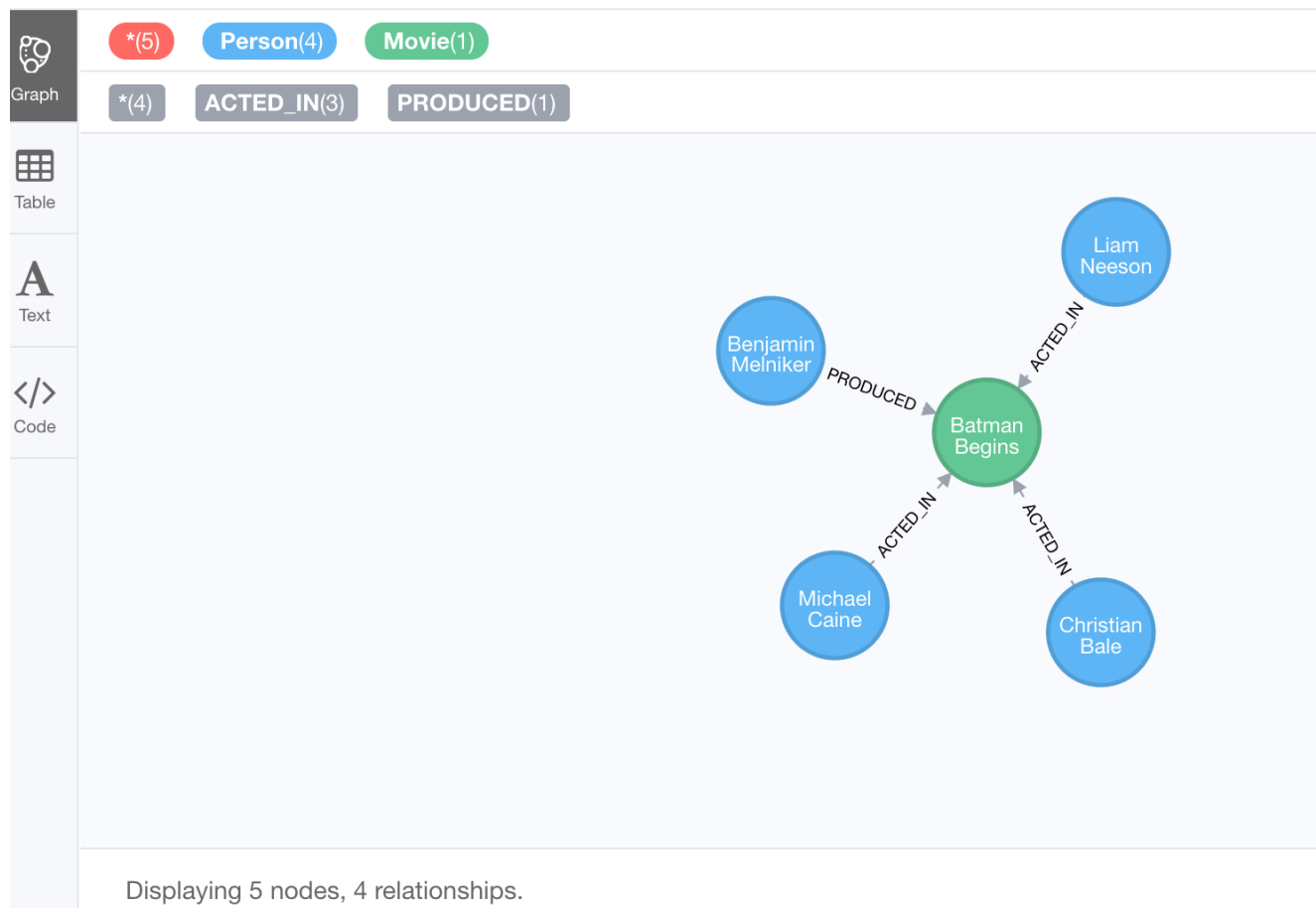
You can delete a relationship between nodes by first finding it in the graph and then deleting it.

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

```
MATCH (a:Person)-[rel:ACTED_IN]->(m:Movie)
WHERE a.name = 'Christian Bale' AND m.title = 'Batman Begins'
DELETE rel
RETURN a, m
```
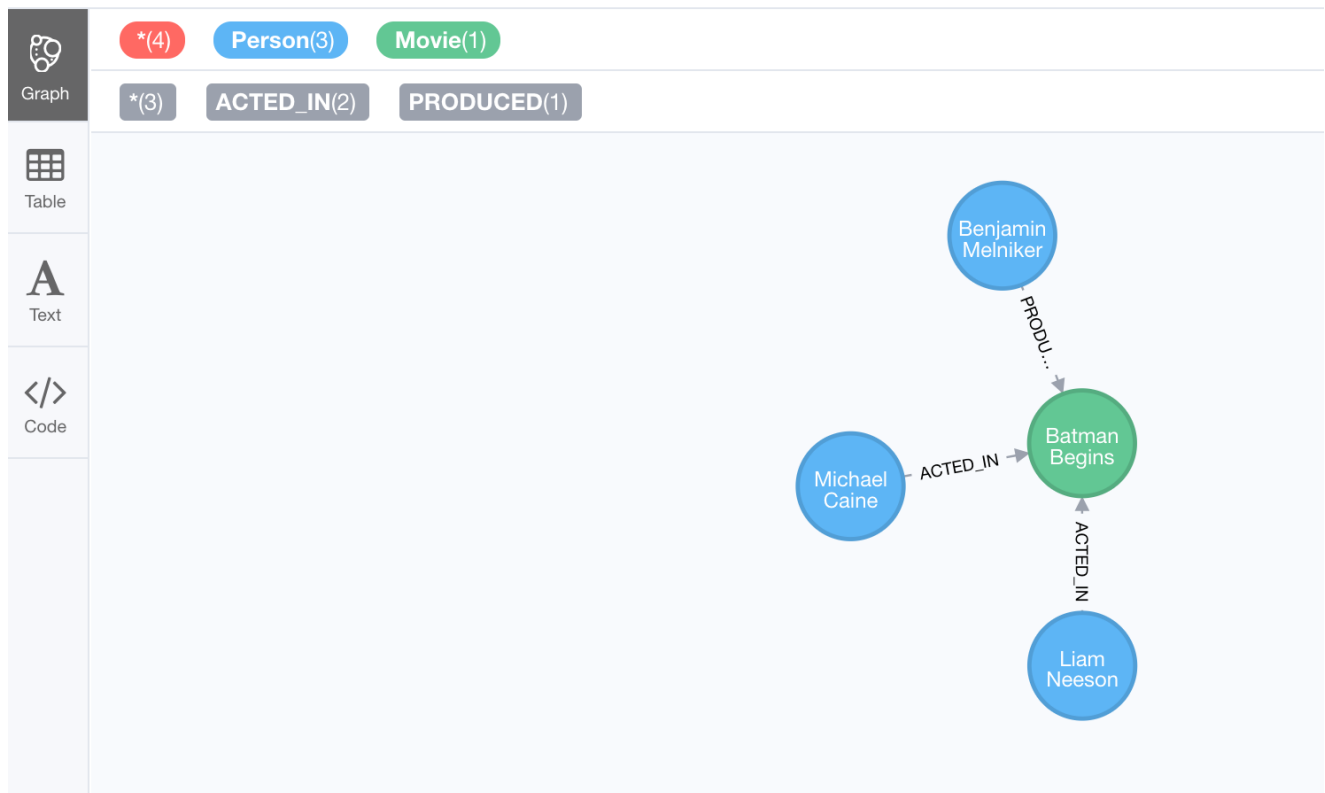
Here is the result of running this Cypher statement:

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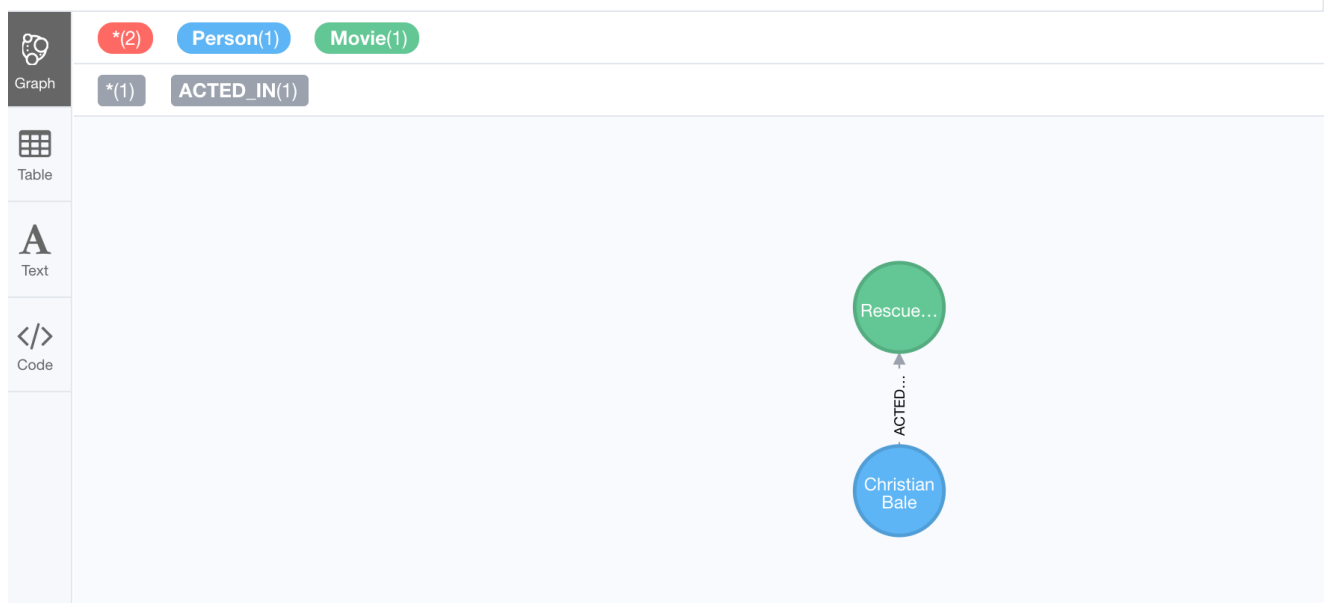
In this example, we find the node for the producer, *Benjamin Melniker*, as well as his relationship to movie nodes. First, we delete the relationship(s), then we delete the node:

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

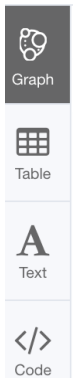Here is the result of running this Cypher statement:

```
$ MATCH (p:Person)-[rel:PRODUCED]->(:Movie) WHERE p.name = 'Benjamin Melniker' DELETE rel, p
```
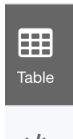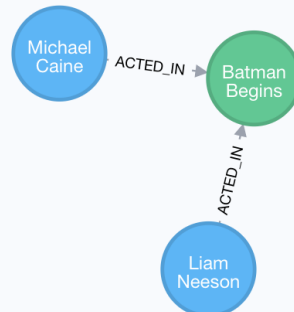
Deleted 1 node, deleted 1 relationship, completed after 3 ms.

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
```


Displaying 3 nodes, 2 relationships.

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:

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

Error

ERROR

Neo.ClientError.Schema.ConstraintValidationFailed

```
Neo.ClientError.Schema.ConstraintValidationFailed: Cannot delete node<1899>, because it still
has relationships. To delete this node, you must first delete its relationships.
```

⚠ Neo.ClientError.Schema.ConstraintValidationFailed: Cannot delete node<1899>, because it still has relationships. To delete this node, you must first...

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
```

Table

Deleted 1 node, deleted 1 relationship, completed after 10 ms.

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
```
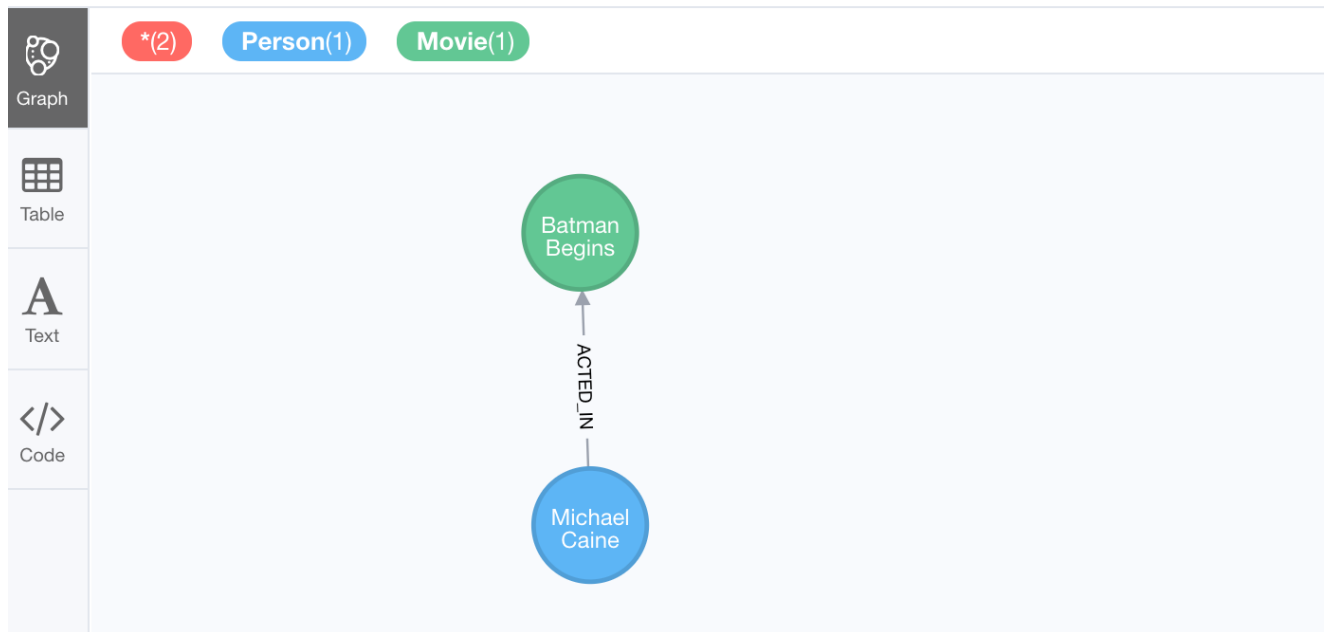

Exercise 10: Deleting Nodes and Relationships

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 existing properties if 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
```

	a
Graph	

Table	{
	    "name": "Michael Caine",
	    "born": 1933
	}

Text	

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
```

|	a
Graph	

Table	{
	    "name": "Michael Caine",
	    "born": 1933
	}

Text	

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
```
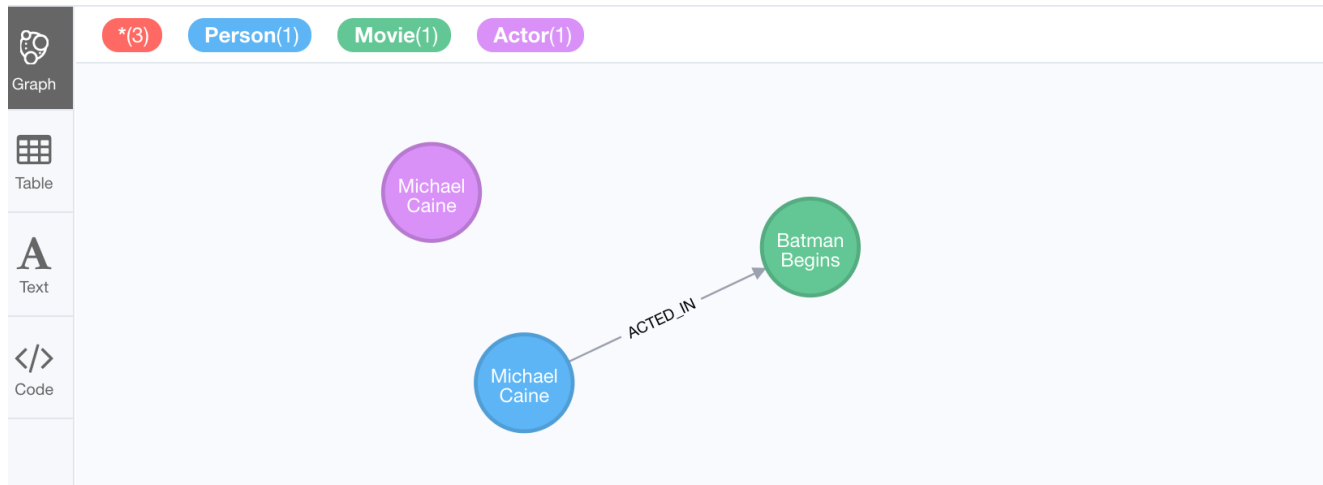

NOTE	Be mindful that node labels and the properties for a node are significant when merging nodes.

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 <u>must</u> 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 it 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.
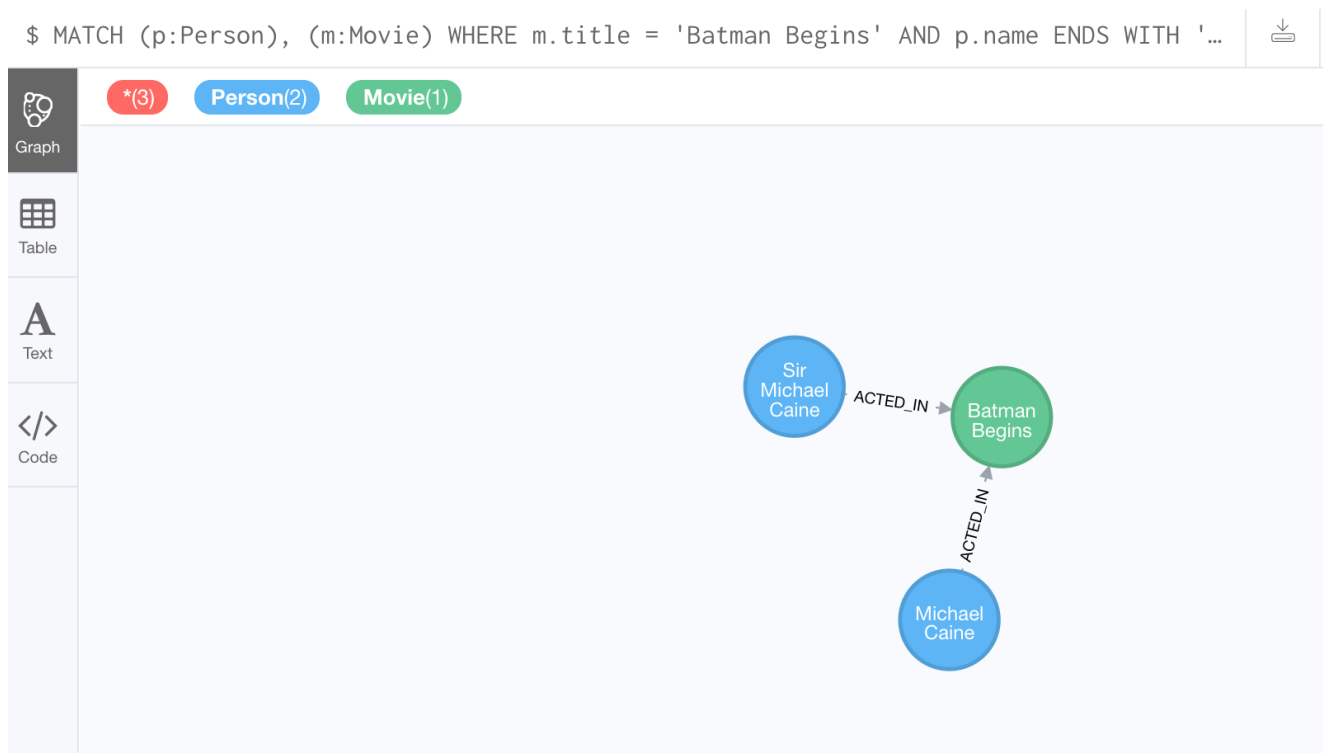
You must be aware of the behavior of the `MERGE` clause and how it will automatically create nodes and relationships. `MERGE` tries to find a full pattern and if it doesn't find it, it creates that full pattern. That's why in most cases you should first `MERGE` your nodes and then your relationship afterwards.

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.

Check your understanding

Question 1

What Cypher clauses can you use to create a node?

Select the correct answers.

- ☐ CREATE
- ☐ CREATE NODE
- ☐ MERGE
- ☐ ADD

Question 2

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

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

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

Select the correct answers.

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

Question 3

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

Select the correct answer.

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

Summary

You should now be able to write Cypher statements to:

- Create a node
  - Add and remove node labels
  - Add and remove node properties
  - Update properties
- Create a relationship
  - Add and remove properties for a relationship
- Delete a node
- Delete a relationship
- Merge data in a graph
  - Creating nodes
  - Creating relationships

Grade Quiz and Continue