Getting More Out of Queries

Table of Contents

About this module

You have learned how to query nodes and relationships in a graph using simple patterns. You learned how to use node labels, relationship types, and properties to filter your queries. Cypher provides a rich set of `MATCH` clauses and keywords you can use to get more out of your queries.

In this module, you will learn how to specify a query criteria using a `WHERE` clause. Then you will learn how to control query processing using multiple `MATCH` clauses, varying length patterns, optional results, collecting results into lists and counting results. Then, you will learn how to limit the number of results returned, eliminate duplication, and ordering the results returned. Finally, you will learn how to work with Cypher lists and date values.

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

- Filter queries using the `WHERE` clause
- Control query processing
- Control what results are returned
- Work with Cypher lists and dates

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 is fixed, where you must specify values for the properties. 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. 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:

You can specify conditions in a `WHERE` clause that return a value of `true` or `f` 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 *Developer Manual's* section *Mathematical Functions* for more information.

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

Testing labels and relationship types

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)-[rel]->(m)
WHERE p:Person AND type(rel) = 'ACTED_IN' 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:

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

Table | Text | Code

m.title	m.tagline
"A Few Good Men"	"In the heart of the nation's capital, in a courthouse of the U.S. government, one man will stop at nothing to keep his honor, and one will stop at nothing to find the truth."
"As Good as It Gets"	"A comedy from the heart that goes for the throat."
"Hoffa"	"He didn't want law. He wanted justice."
"One Flew Over the Cuckoo's Nest"	"If he's crazy, what does that make you?"

Note that there is a difference between a property not existing and a property having a `null` value. Cypher has keywords that you can use to test if a property value is `null`. You can use `IS NULL` and `IS NOT NULL` for these types of tests.

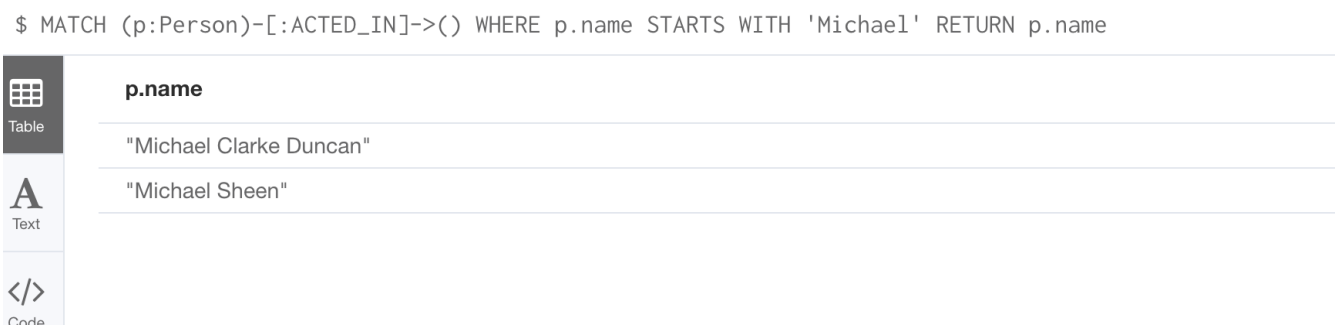
Testing strings

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
```

See the *String functions* section of the *Developer 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 against. Here is an example where we test the value 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:

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

Table | Text | Code

p.name
"Tom Cruise"
"Tom Skerritt"
"Tom Hanks"
"Tom Tykwer"

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 di d movies:

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

Here is the result:

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

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

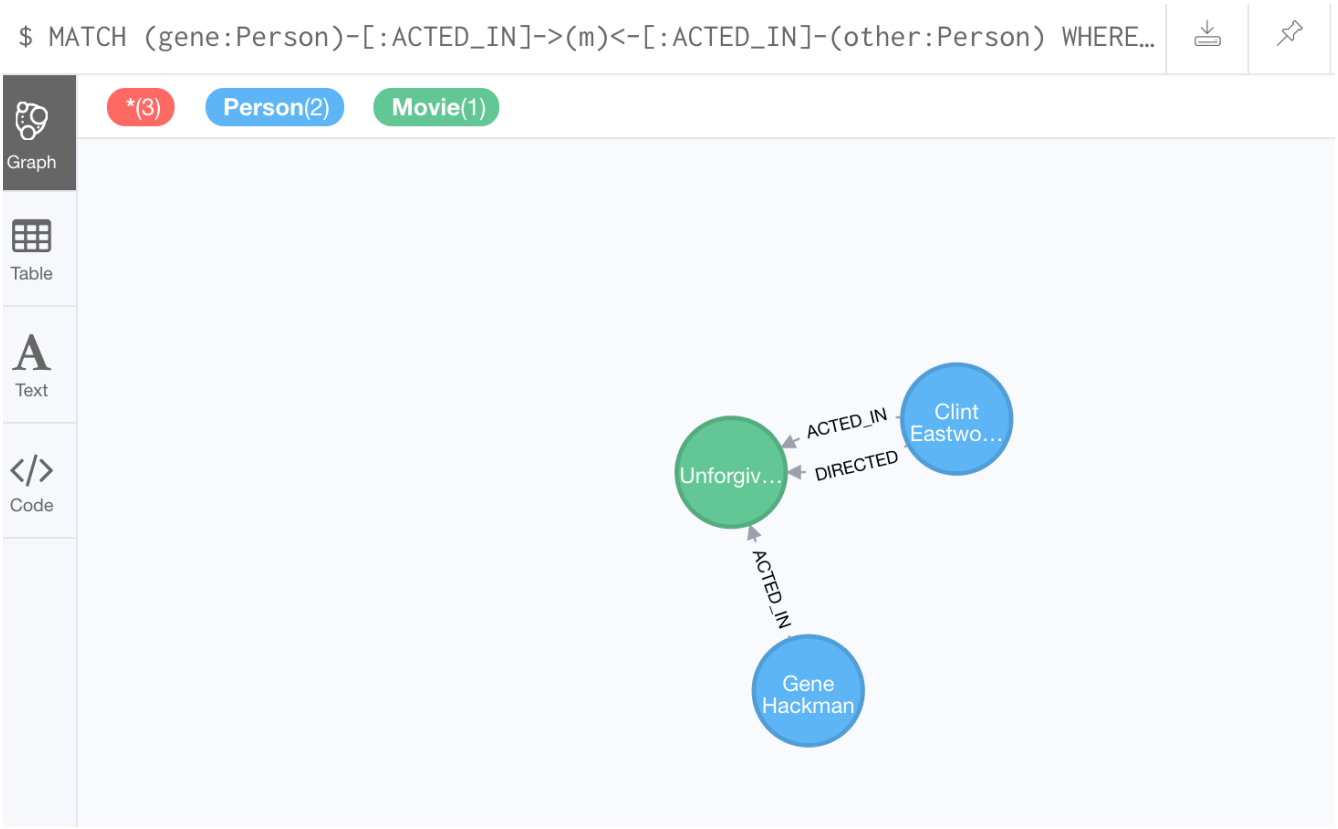
Started streaming 6 records in less than 1 ms and completed after 1 ms.

Another type of test you can perform against a pattern is if the pattern exists. Earlier, you learned how to check the existence of a property. You can also check the existence of a particular path in the graph.

Here is an example where we want to find *Gene Hackman* and the movie that he acted in with another person who also directed the movie. We use `exists()` to further qualify the relationship of the other person.

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

Here is the result:



Testing against list values

If you have a set of values you want to test against, you can place them in a list or you can test against 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 against 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 against 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
```

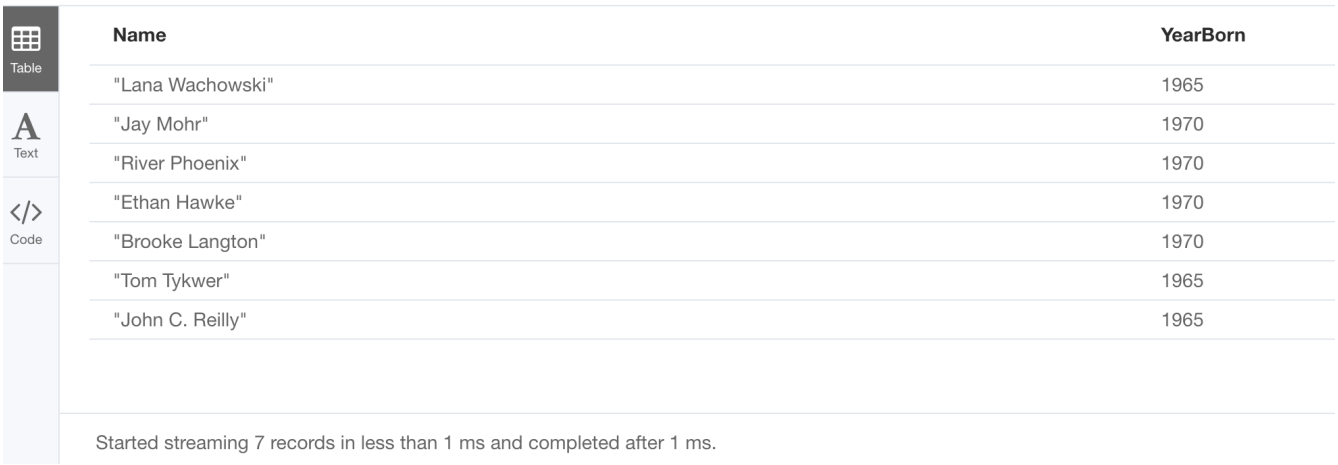


Table | Text | Code

Name	YearBorn
"Lana Wachowski"	1965
"Jay Mohr"	1970
"River Phoenix"	1970
"Ethan Hawke"	1970
"Brooke Langton"	1970
"Tom Tykwer"	1965
"John C. Reilly"	1965

Started streaming 7 records in less than 1 ms and completed after 1 ms.

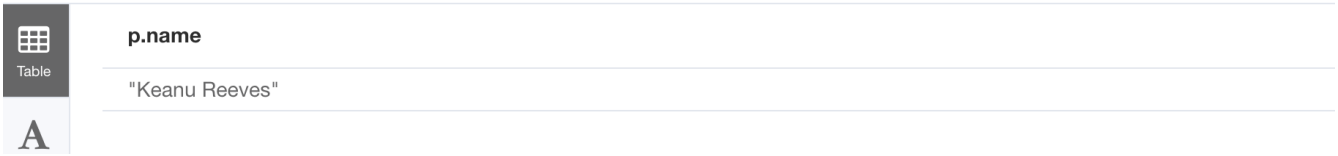
You can also compare a value to an existing list in the graph.

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
```



Table

p.name
"Keanu Reeves"

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

Exercise 4: Filtering queries using the `WHERE` clause

In the query edit pane of Neo4j Browser, execute the browser command: `:play http://guides.neo4j.com/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

A best practice is to create your queries using a single MATCH pattern. This MATCH clause includes two patterns separated by a co :

```
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:

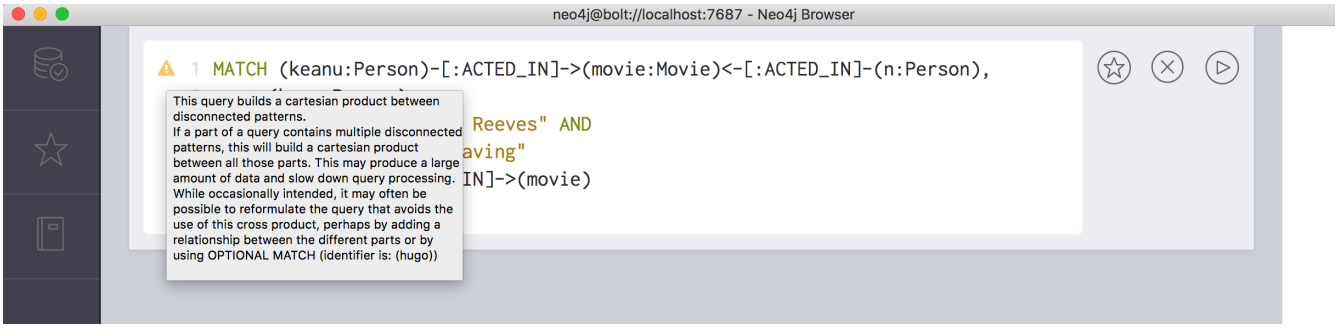
```
MATCH (a:Person)-[:ACTED_IN]->(m:Movie)<-[:DIRECTED]-(d:Person)
WHERE m.released = 2007
RETURN a.name, m.title, d.name
```

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.

Here are some examples of specifying two patterns 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 patterns 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:

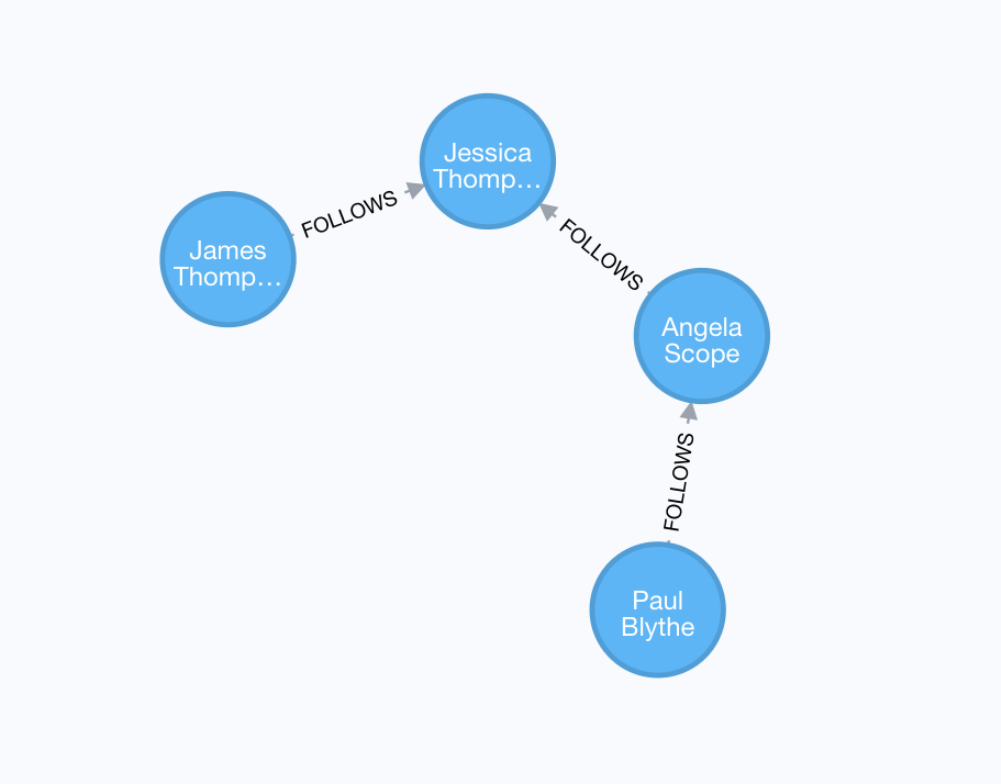
Here is another example where two patterns are necessary. Suppose we want to retrieve the movies that *Meg Ryan* acted in and their respective directors, as well as the other actors that acted in these movies. Here is the query to do this:

```
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:

Specifying varying length patterns

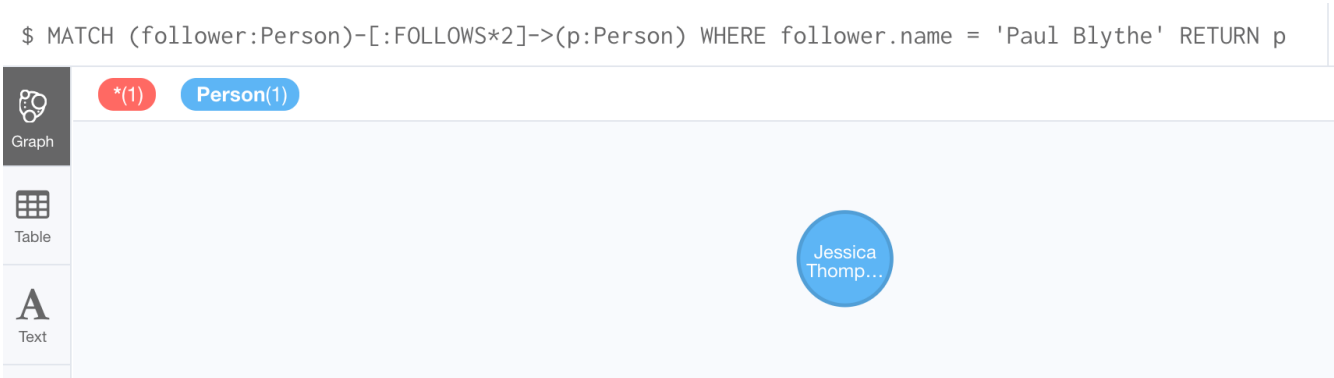
Any graph that represents social networking will most likely have multiple paths of varying lengths. Think of the *friend* relationship in *Face* and how connections are made by friends of friends, etc. The *Movie* database for this training does not have much depth of relationships, but it does have 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 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 is the simplified syntax for how varying length patterns are specified in Cypher:

(*node1*)-[`:RELTYPE` *]→(*node2*)	retrieve all paths with the relationship, `RELTYPE` from *node1* to *node2* and beyond.

(*node1*)-[:RELTYPE *]-(*node2*)	retrieve all paths with the relationship, RELTYPE from *node1* to *node2* or from *node2* to *node1* and beyond.
(*node1*)-[:RELTYPE *3]→(*node2*)	retrieve the paths of length 3 with the relationship, RELTYPE from *node1* to *node2* and 2 more hops from *node2*.
(*node1*)-[:RELTYPE *1..3]→(*node2*)	retrieve the paths of lengths 1, 2, or 3 with the relationship, RELTYPE from *node1* to *node2*, *node2* to *node3*, as well as, *node3* to (node4) (up to three hops).

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

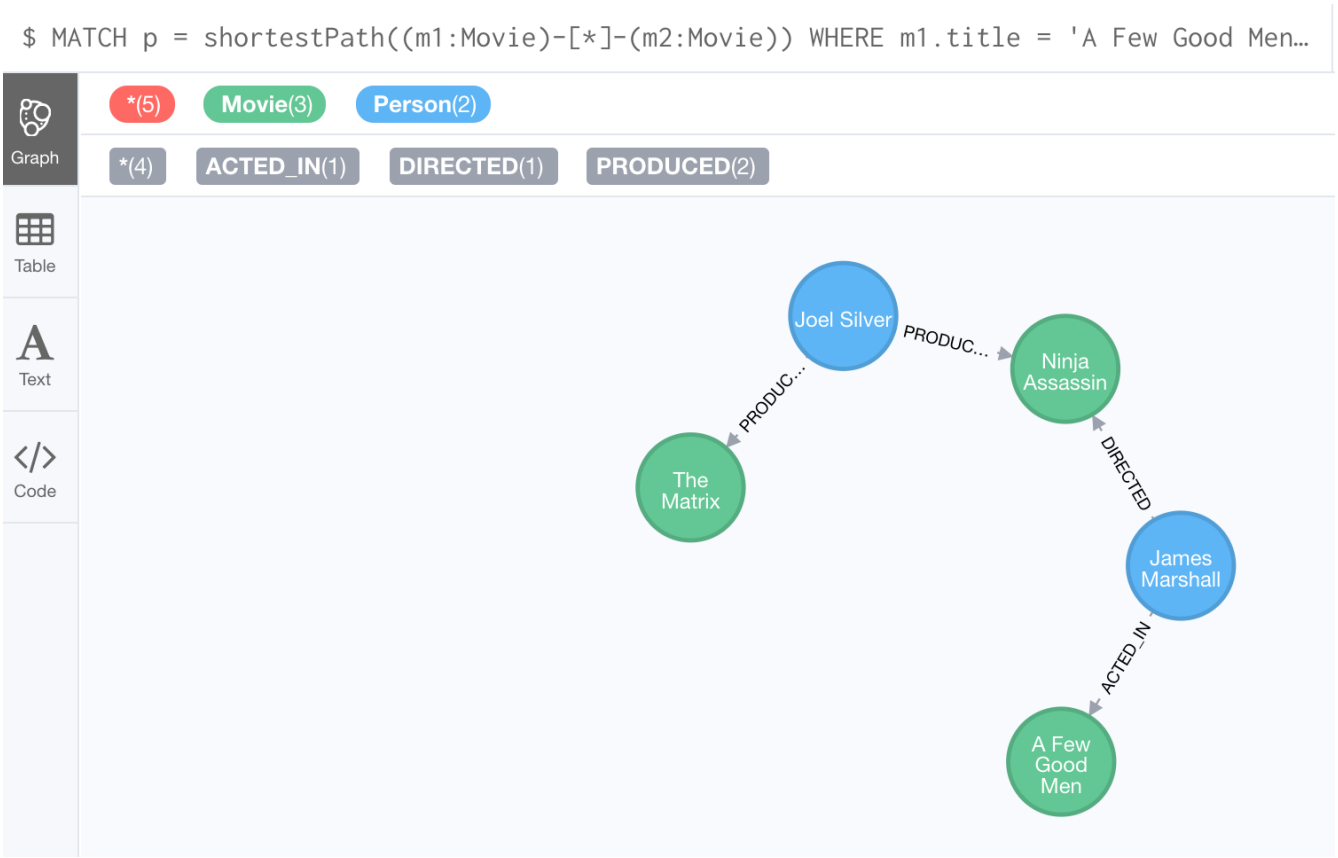
Finding the shortest path

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.

Specifying optional ~~resul~~

`OPTIONAL MATCH` matches patterns against 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)-[:REVIEWED]->(m:Movie)
RETURN p.name, m.title
```

Here is the result returned:

```
$ MATCH (p:Person) WHERE p.name STARTS WITH 'James' OPTIONAL MATCH (p)-[:REVIEWED]->(m:Movie) RETURN
```

Table | Text | Code

p.name	m.title
"James Marshall"	null
"James L. Brooks"	null
"James Cromwell"	null
"James Thompson"	"The Da Vinci Code"
"James Thompson"	"The Replacements"

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

Collecting re

Cypher has a built-in function, `collect()` that enables you to place a result 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 `Tom Cruise Movies`
```

Here is the result returned:

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

Table | Text

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

Counting results

The Cypher `count()` function is very useful whe want to count the number of occurrences of a particular query result. If you specify `count` the graph engine calculates the number of occurrences of *x*. If you specify `count(*)`, the graph engine calculates the number of rows retrieved, including those with `null` values.

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:

```
$ MATCH (actor:Person)-[:ACTED_IN]->(m:Movie)<-[:DIRECTED]-(director:Person) RETURN actor.name, director.name, count(m) AS…
```

Table | Text | Code

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"]
"Hugo Weaving"	"Lana Wachowski"	4	["The Matrix", "The Matrix Reloaded", "The Matrix Revolutions", "Cloud Atlas"]
"Laurence Fishburne"	"Lana Wachowski"	3	["The Matrix", "The Matrix Reloaded", "The Matrix Revolutions"]
"Jay Mohr"	"Cameron Crowe"	1	["Jerry Maguire"]
"Hugo Weaving"	"James Marshall"	1	["V for Vendetta"]

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 *Developer 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.

Here is an example where we start the query processing by retrieving all actors and their movies. During the query processing, we want to count the number of movies per actor, as well as collect the titles for the movies, ~~but we want to stop the retrieval for an actor's movies once we have retrieved at most three movies.~~ 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:

```
$ MATCH (a:Person)-[:ACTED_IN]->(m:Movie) WITH a, count(a) AS NumMovies, collect(m.title) as Movies WHERE NumMovies > 1 AN…
```

Table | Text | Code

a.name	NumMovies	Movies
"Ben Miles"	3	["V for Vendetta", "Speed Racer", "Ninja Assassin"]
"J.T. Walsh"	2	["A Few Good Men", "Hoffa"]
"Robin Williams"	3	["What Dreams May Come", "The Birdcage", "Bicentennial Man"]
"Bonnie Hunt"	2	["Jerry Maguire", "The Green Mile"]
"Marshall Bell"	2	["Stand By Me", "RescueDawn"]
"Greg Kinnear"	2	["As Good as It Gets", "You've Got Mail"]
"Charlize Theron"	2	["The Devil's Advocate", "That Thing You Do"]
"Laurence Fishburne"	3	["The Matrix", "The Matrix Reloaded", "The Matrix Revolutions"]
"Oliver Platt"	2	["Frost/Nixon", "Bicentennial Man"]
"Nathan Lane"	2	["Joe Versus the Volcano", "The Birdcage"]
"Helen Hunt"	3	["As Good as It Gets", "Twister", "Cast Away"]
"Gary Sinise"	2	["The Green Mile", "Apollo 13"]
"Gene Hackman"	3	["The Replacements", "The Birdcage", "Unforgiven"]
"Carrie-Anne Moss"	3	["The Matrix", "The Matrix Reloaded", "The Matrix Revolutions"]

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 lat the query. Notice that for the `RETURN` clause, *a*, *NumMovies*, and *Movies* are available for use.

Here is another example where we want to find all actors who have acted in at least 5 movies, and possibly directed the movie.

```
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:

```
$ MATCH (p:Person) WITH p, size((p)-[:ACTED_IN]->(:Movie)) AS mo…
```

Table | Text | Code

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

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 5, 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 5 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 http://guides.neo4j.com/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:

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

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", "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"]
2004	["The Polar Express"]
1995	["Apollo 13"]

Started streaming 12 records after 1 ms and completed after 1 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, collect(DISTINCT m.title)…
```

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"]
2004	["The Polar Express"]
1995	["Apollo 13"]

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 property 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.

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, collect(DISTINCT m.title)…
```

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"]
2004	["The Polar Express"]
1995	["Apollo 13"]

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

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:

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

Table | Text | Code

Title	Year
"Cloud Atlas"	2012
"Ninja Assassin"	2009
"Frost/Nixon"	2008
"Speed Racer"	2008
"Charlie Wilson's War"	2007
"V for Vendetta"	2006
"RescueDawn"	2006
"The Da Vinci Code"	2006
"The Polar Express"	2004
"The Matrix Revolutions"	2003

Started streaming 10 records after 1 ms and completed after 2 ms.

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 also terminate a query once it has reached the desired number of results.

In this example, we count the number of movies during the query and we want to stop the processing of the movies for an actor 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:

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

Table | Text | Code

a.name	NumMovies	Movies
"Jack Nicholson"	5	["A Few Good Men", "As Good as It Gets", "Hoffa", "One Flew Over the Cuckoo's Nest", "Something's Gotta Give"]
"Meg Ryan"	5	["Top Gun", "You've Got Mail", "Sleepless in Seattle", "Joe Versus the Volcano", "When Harry Met Sally"]
"Hugo Weaving"	5	["The Matrix", "The Matrix Reloaded", "The Matrix Revolutions", "Cloud Atlas", "V for Vendetta"]

Exercise 6: Controlling results returned

In the query edit pane of Neo4j Browser, execute the browser command: `:play http://guides.neo4j.com/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 used to build or access elements in lists. A Cypher `map` is a ~~special type of list where each~~ element is a key/value pair.

For example, 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 *Developer Manual*.

Unwinding lists

Earlier you learned how you can create lists from results during the query. There may be some situations where you want to perform the opposite of collecting results, but rather separate the results 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 code, but you do need the `RETURN` statement here to return the calculated values from the Cypher code.

Note: The `UNWIND` clause is frequently used when importing ~~relational~~ data into a graph.

Dates

Cypher has a built-in `date()` function you can use to calculate dates. You use a combination of numeric 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 the Cypher code to retrieve all actors from the graph, and if they have a value for *born*, calculate the *age* value. Notice that the `date()` function is called and then converted to a string and then an integer to perform the age calculation.

```
MATCH (actor:Person)-[:ACTED_IN]->(:Movie)
// calculate the age
with actor, toInteger(left(toString(date.truncate('year', date())), 4)) - actor.born
as age
WHERE exists(actor.born)
RETURN actor.name, age as `Age today`
      ORDER BY actor.born DESC
```

date().year

Here is the result returned:

```
$ MATCH (actor:Person)-[:ACTED_IN]->(:Movie) with actor, …
```

Table | Text | Code

actor.name	Age today
"Jonathan Lipnicki"	22
"Emile Hirsch"	33
"Rain"	36
"Rain"	36
"Natalie Portman"	37
"Christina Ricci"	38
"Emil Eifrem"	40
"Liv Tyler"	41
"Audrey Tautou"	42
"Charlize Theron"	43
"Charlize Theron"	43
"Jerry O'Connell"	44
"Jerry O'Connell"	44
"Christian Bale"	44

Consult the *Developer 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 http://guides.neo4j.com/intro-neo4j-exercises` and follow the instructions for Exercise 7.

Check your understanding

Question 1

Suppose you want to add a `WHERE` clause at the end of this code to filter the results retrieved.

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

What variables, can you test in the `WHERE` clause:

Select the correct answers.

- ☐ p
- ☐ rel
- ☐ m
- ☐ PRODUCED

Question 2

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

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

What keyword do you specify for XX?

Select the correct answer.

- ☐ CONTAINS
- ☐ IN
- ☐ IS
- ☐ EQUALS

Question 3

Given this Cypher query:

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

What variables or aliases can be used to return values?

Select the correct answers.

- ☐ a
- ☐ m
- ☐ NumMovies
- ☐ Movies

Summary

You should now be able to write Cypher code to:

- Filter queries using the `WHERE` clause
- Control query processing
- Control what results are returned
- Work with Cypher lists and dates