COMP3008 - Big Data Analytics

Workshop 4 - Recommender Systems

Collaborative and content-based filtering

This practical constitutes our first approach to recommender systemscomputer systems that make recommendations are known as recommender systems. We will learn how to use the relationships in a dataset to identify potential connections that can be derived from the relationships in the graph.

In exercise 1, the practical employs a small collection of actors (133 in total), which are taken from a movie database. By following the relationships between *actors* and *movies*, you can determine occurrences of actors working together, the frequency of actors working with one another, and the movies they have produced in common. This is basic information that we can later exploit in the context of movie recommendations.

In exercise 2, we will create a friendship graph for some of the characters in “Romeo and Juliet” and will be implementing the collaborative filtering algorithm to identify friend recommendations for the characters appearing in the graph above. Note that the practical employs a small collection of characters (11 in total) to facilitate the testing of different options on a small dataset.

Exercise 1 - Movie Database

Uploading the Dataset

Rather than importing the dataset from a CSV file, as we have done it in previous practical sessions, we will populate our database using a series of commands listed in the file `movie_dataset_script.txt`, which is available on the DLE. The `movie_dataset_script.txt` comprises multiple clauses, which we can use to create our database.

Once you have a running instance of Neo4j on your screen, copy the entire contents of the `movie_dataset_script.txt` file and paste them into the Neo4j Browser.

Click on the *play* button (*cypher run button*). Provided you execute the commands in the `movie_dataset_script.txt` file correctly, you should be able to create 133 actors, 38 movies and 253 relationships among actors and moviesthe relationships have different labels.

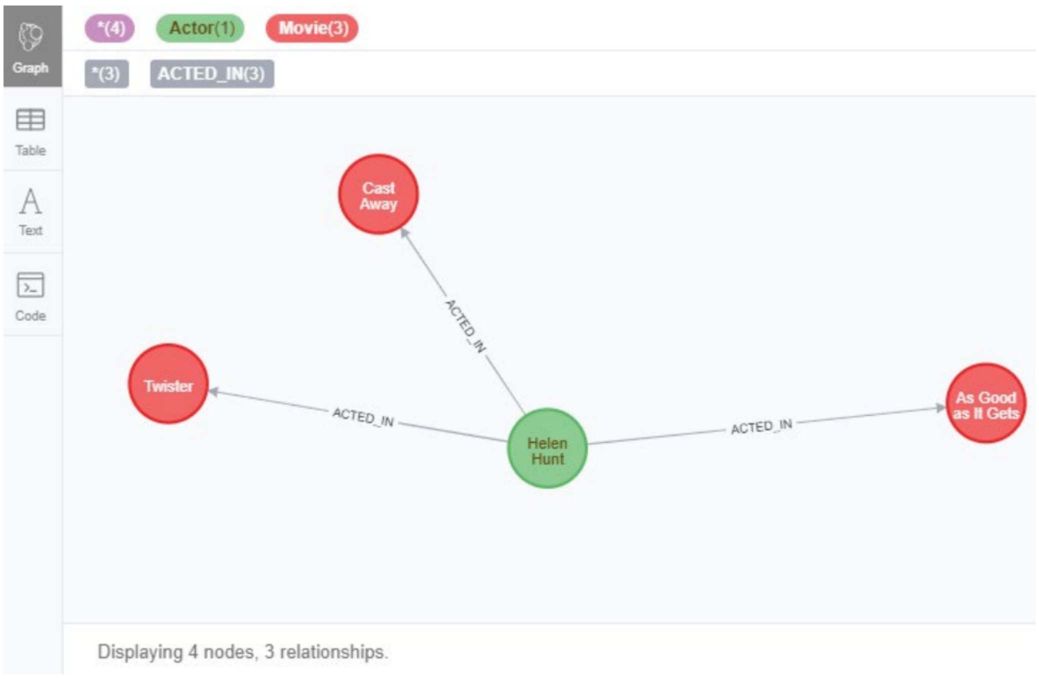
Familiarise yourself with the dataset and the relationships you have created. For example, count the total number of *Actor nodes* (133) and *Movie nodes* (38). You can count the number of Actor nodes by running the following command:

```
MATCH (a:Actor)
RETURN COUNT(a) as ACTORS
```

Count the total number of DIRECTED relationships (44), ACTED_IN relationships (172), PRODUCED relationships (15), FOLLOWS relationships (3), REVIEWED relationships (9), and WROTE relationships (10). You can count the number of DIRECTED relationships by running the following command:

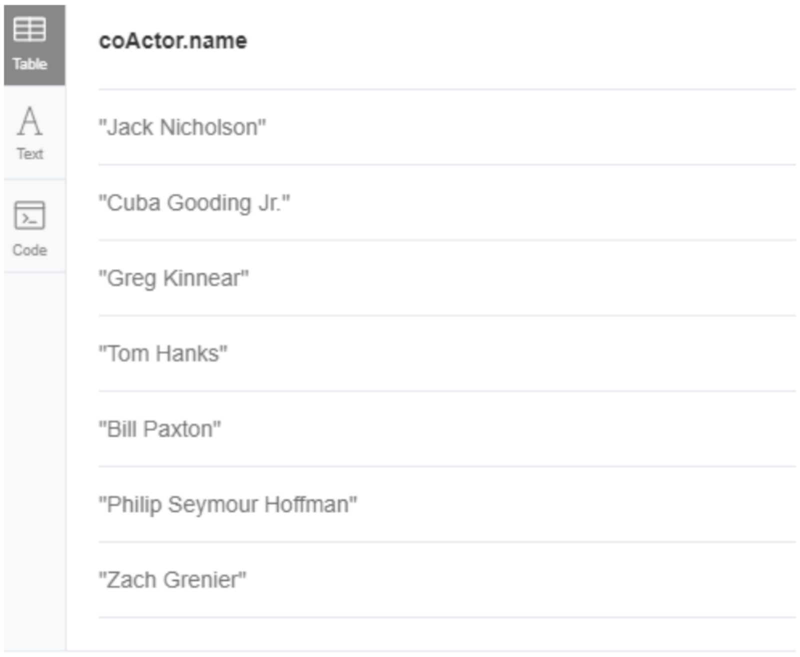
```
   MATCH () - [r:DIRECTED]→ ()
RETURN COUNT(r) as DIRECTED_MOVIES
```

Find all the movies where Helen Hunt acts in. You can achieve this by finding the node corresponding to Helen Hunt and then following all the ACTED_IN relationships starting from Helen Hunt node. Look for a way to show the results visually in a graph, rather than as text. Your results should look similar to the following picture.



Displaying 4 nodes, 3 relationships.

Find all the actors who have acted with Helen Hunt in her movies. In other words, find the collection of Helen Hunts *co-actors*. You should get exactly 7 actorsthe names are listed below.

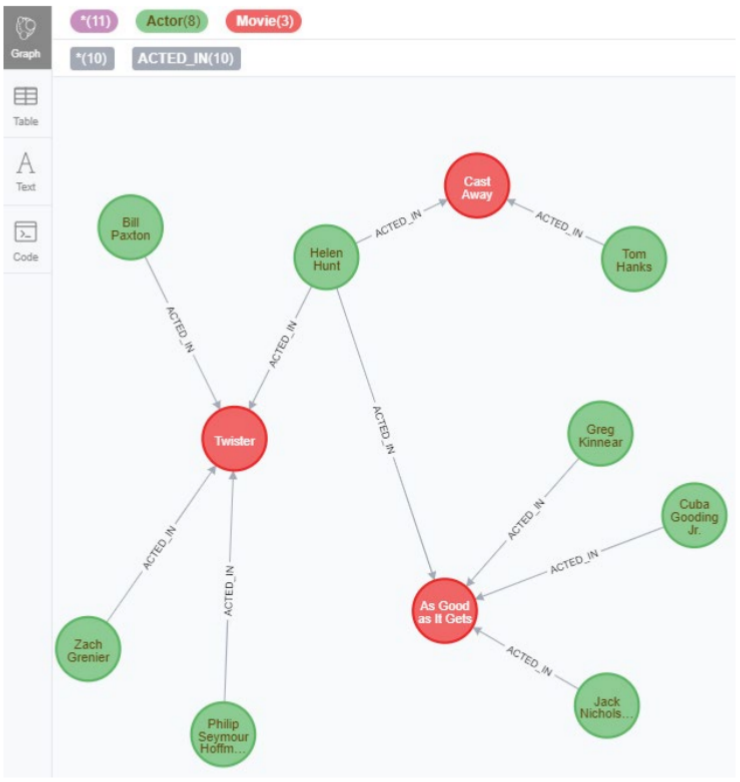


coActor.name
"Jack Nicholson"
"Cuba Gooding Jr."
"Greg Kinnear"
"Tom Hanks"
"Bill Paxton"
"Philip Seymour Hoffman"
"Zach Grenier"

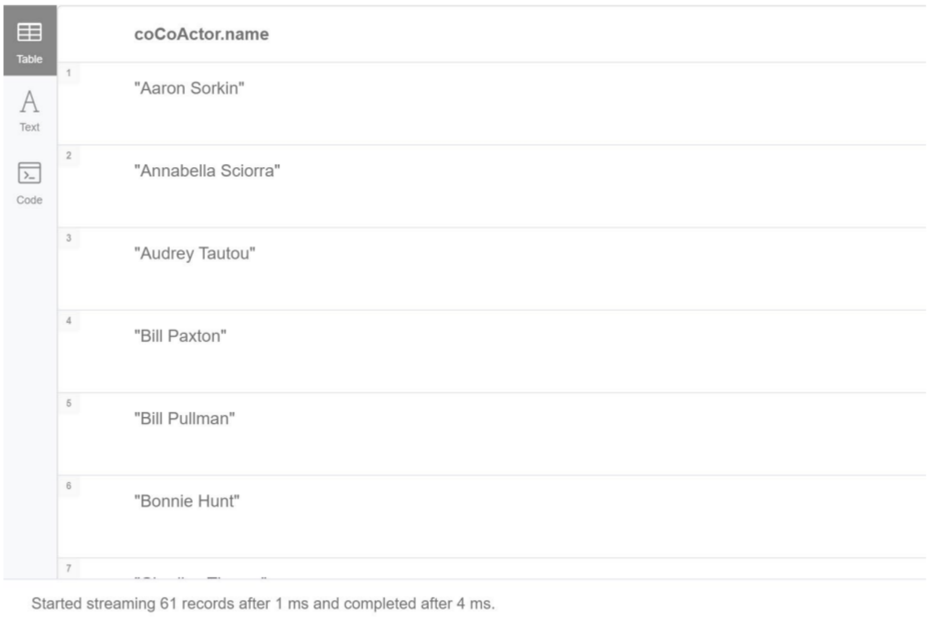
Started streaming 7 records in less than 1 ms and completed in less than 1 ms.

Display the collection of Helen Hunts co-actors in a graph, and the movies where they have acted together with

Helen Hunt. Note that we are not interested in a list showing names as textwe want to produce a graph. Your results should be like the following picture.



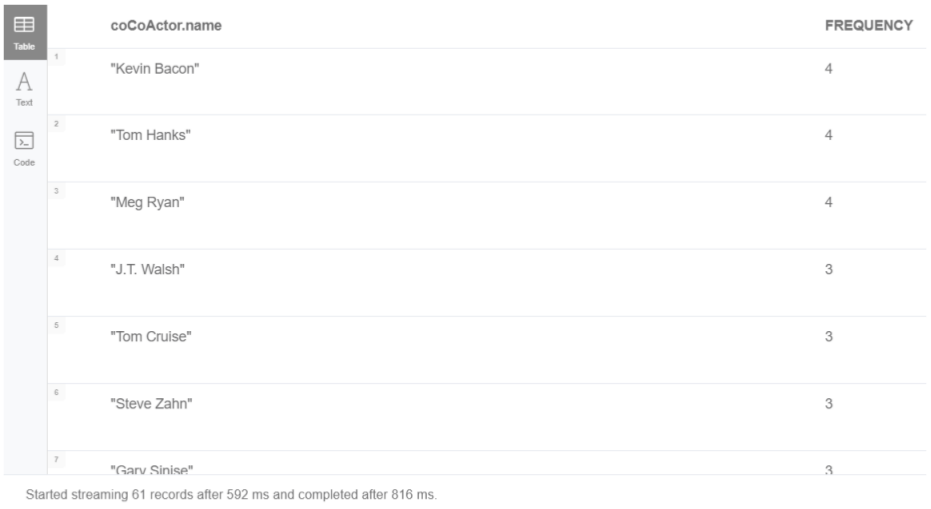
Once you have identified Helen Hunts co-actors, look for Helen Hunts *co-coactors*. These are all the actors who have acted in a movie with a co-actor of Helen Hunt, but have not yet acted in a movie with Helen Hunt. You will find a long list of actors (61 records in total)see the following picture showing the top of the list.



	coCoActor.name
1	"Aaron Sorkin"
2	"Annabella Sciorra"
3	"Audrey Tautou"
4	"Bill Paxton"
5	"Bill Pullman"
6	"Bonnie Hunt"
7	

Started streaming 61 records after 1 ms and completed after 4 ms.

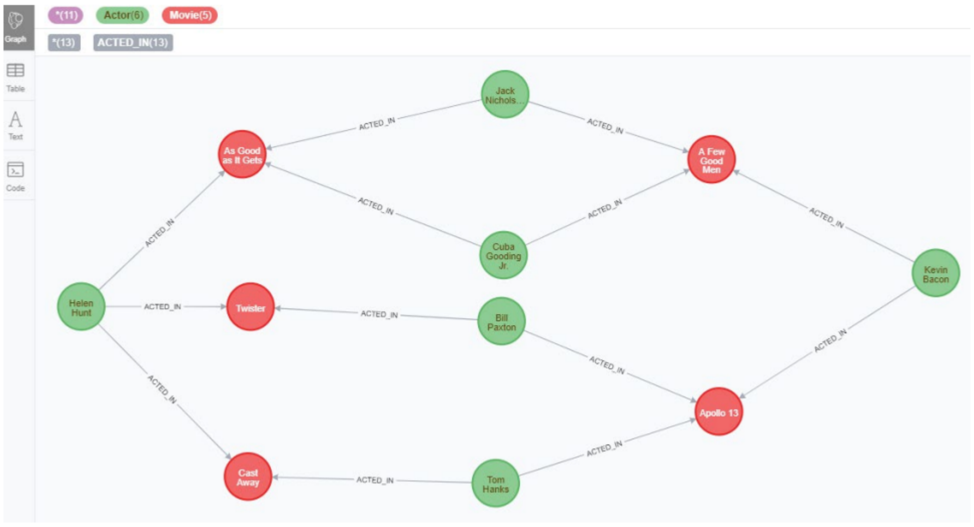
Produce a query to identify which co-co-actors appear most often in Helen Hunts network. This can be achieved by taking the frequency of occurrences into account. This means counting the number of paths between Helen Hunt and each co-co-actor, and ordering them by the highest to the lowest value. The top actors in this list are displayed below.



	coCoActor.name	FREQUENCY
1	"Kevin Bacon"	4
2	"Tom Hanks"	4
3	"Meg Ryan"	4
4	"J.T. Walsh"	3
5	"Tom Cruise"	3
6	"Steve Zahn"	3
7	"Gary Sinise"	3

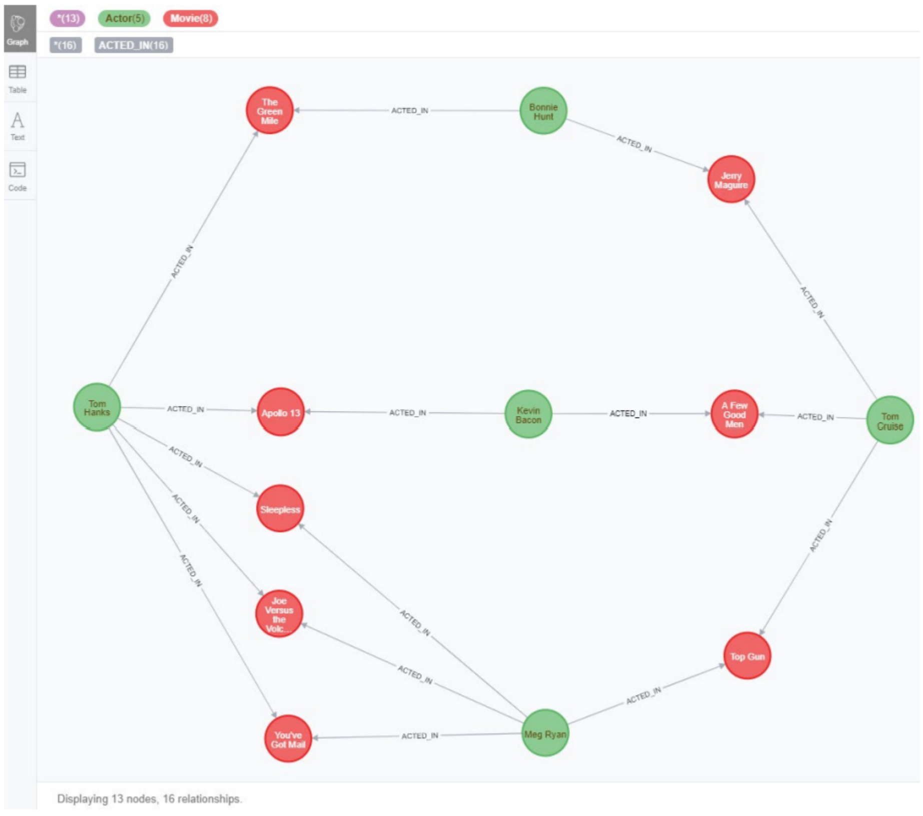
Started streaming 61 records after 592 ms and completed after 816 ms.

The first actor in the list above is Kevin Bacon. Look for all the movies and actors that are located between Helen Hunt and Kevin Bacon. Your query should produce the following results:



As you can see above, there are multiple paths between Helen Hunt and Kevin Bacon. If you have ever played the six degrees of Kevin Bacon game, you will recognise the idea of looking for the number of hops between people. This is exactly what the graph above depicts.

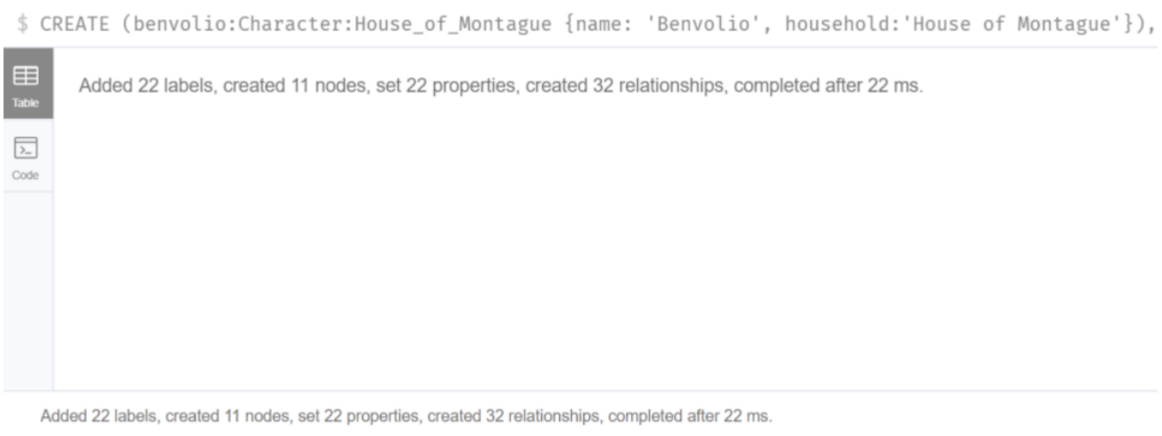
Finally, identify all the movies and actors located in the graph between Tom Hanks and Tom Cruise.



Exercise 2 - Romeo and Juliet

You will create and simultaneously populate your database using a series of commands listed in the file `romeo_and_juliet_graph.txt`, which is available on the DLE. The file comprises multiple clauses involving the CREATE and MERGE commands. Make sure you understand how CREATE and MERGE work and ask for help if you require clarification on this. Once you have a running instance of Neo4j on your screen, copy the entire contents of the `romeo_and_juliet_graph.txt` file and paste them into the Neo4j Browser.

Click on the play button (*cypher run button*). Provided you copied and pasted the commands in the file correctly, you should be able to create 11 *characters*, 3 of them belonging to the *House of Montague*, 4 to the *House of Capulet*, 3 to the *House of Verona*, and 1 *Franciscan*. You should also create 32 `FRIENDS_WITH` relationships.



```
$ CREATE (benvolio:Character:House_of_Montague {name: 'Benvolio', household:'House of Montague'}),
```

```
Added 22 labels, created 11 nodes, set 22 properties, created 32 relationships, completed after 22 ms.
```

```
Added 22 labels, created 11 nodes, set 22 properties, created 32 relationships, completed after 22 ms.
```

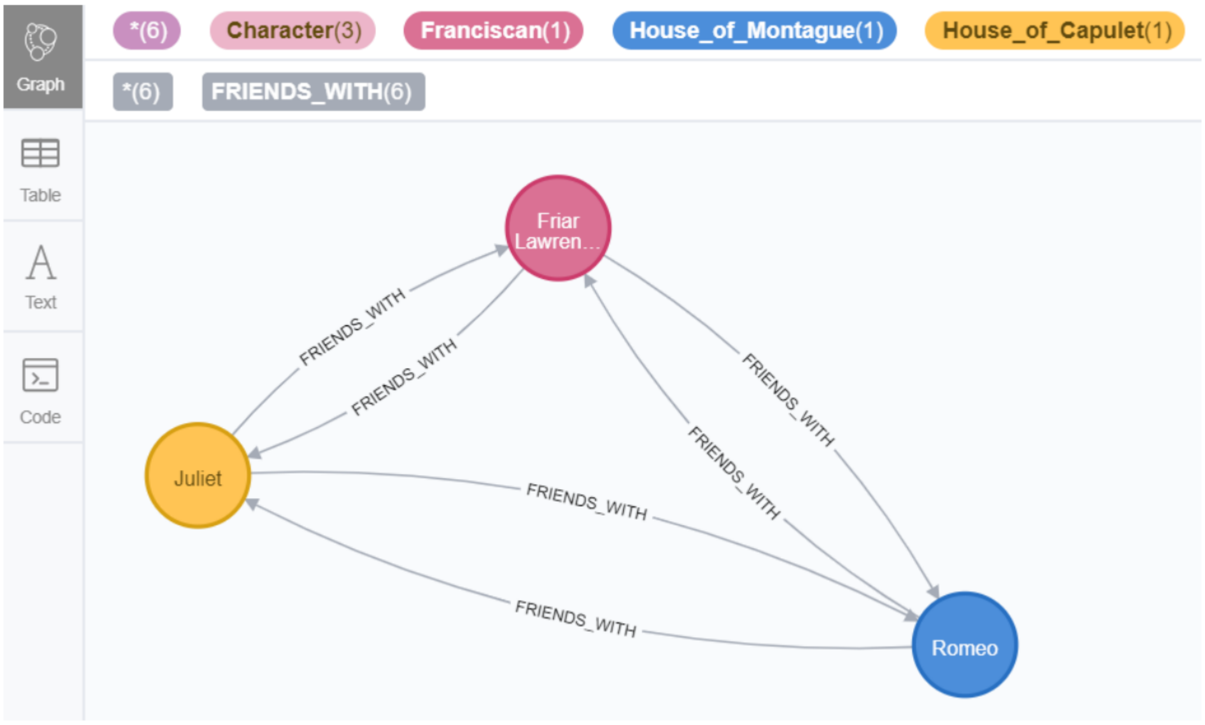
Familiarise yourself with the dataset and the relationships you have created. For example, count the total number of Franciscan characters (1) - you can count this by running the following command:

```
MATCH (n:Character:Franciscans)
RETURN COUNT(n) as TOTAL_NUMBER_OF_FRANCISCANS
```

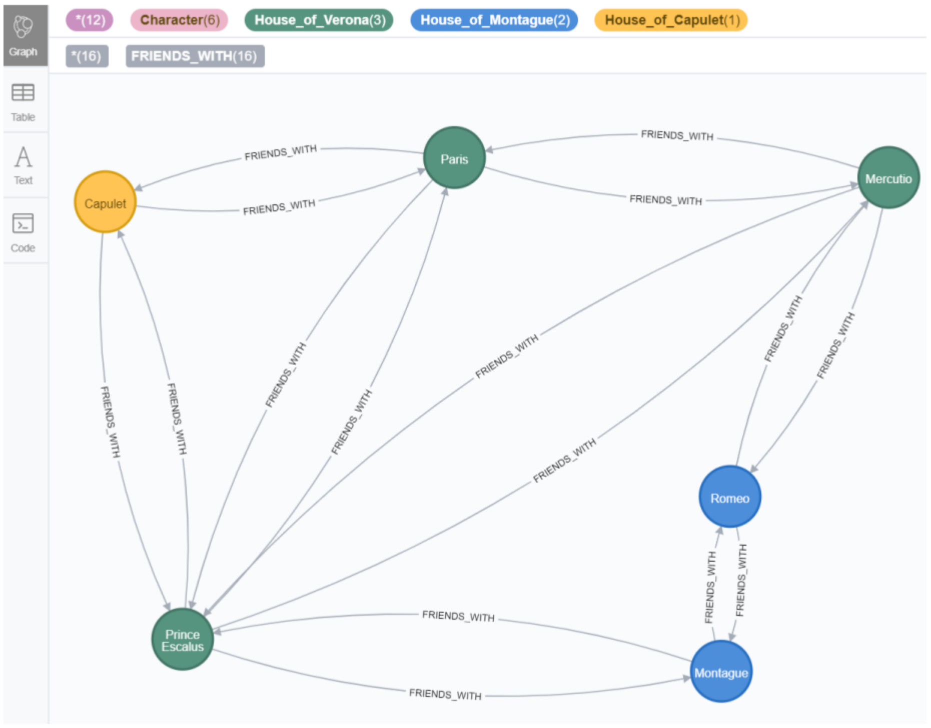
Can you explain why the following command produces the same result? If not, ask for clarification.

```
MATCH (n:Franciscans)
RETURN COUNT(n) as TOTAL_NUMBER_OF_FRANCISCANS
```

Display on the screen a graph showing all the Franciscan characters and all the relationships among them and the other characters in the graph. The results should be similar to the following picture.

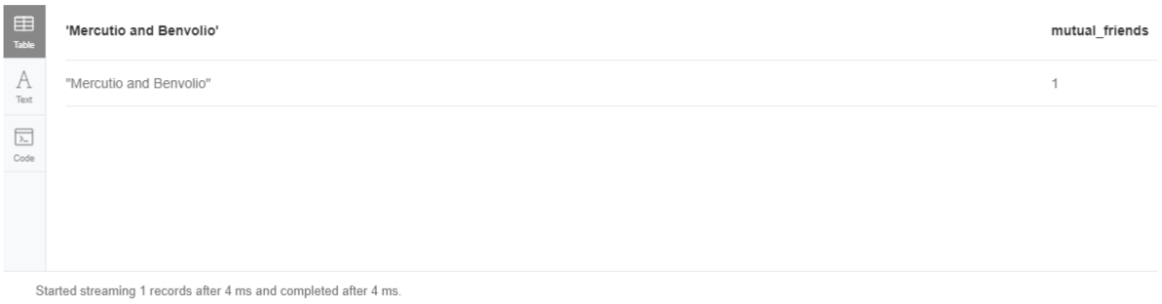


Following the same approach you employed in the previous query, display a graph showing all the characters belonging to the House of Verona, and all the relationships between them and the rest of the characters in the graph. Your results should look like the following picture.



The underlying assumption behind collaborative filtering is that if person A has the same opinion as a person B on a certain issue, A and B are more likely to share an opinion on a different issue than randomly chosen people. For example, if you like John, Paul and George, and another person like John, Paul, George and Ringo, then it stands to reason that you will like Ringo as well.

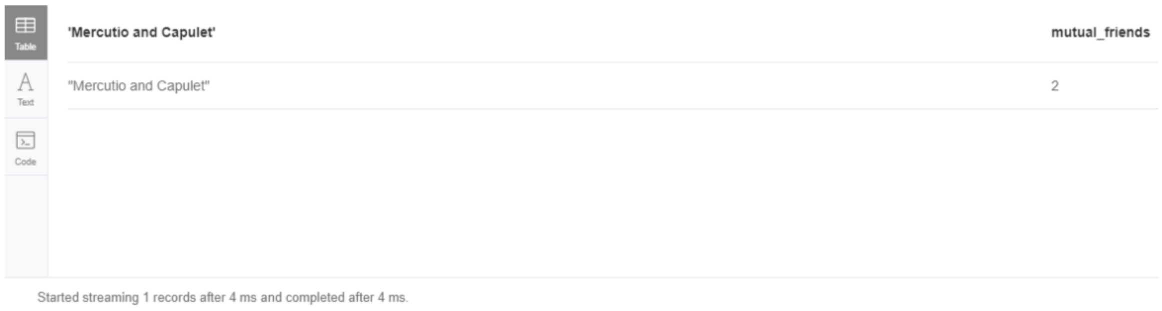
For the purpose of testing this algorithm in the practical session, start by counting the number of mutual friends between Mercutio and Benvolio. Make sure the names Mercutio and Benvolio are followed by the number of mutual friends between them. Your results should look like the following picture.



'Mercutio and Benvolio'	mutual_friends
"Mercutio and Benvolio"	1

Started streaming 1 records after 4 ms and completed after 4 ms.

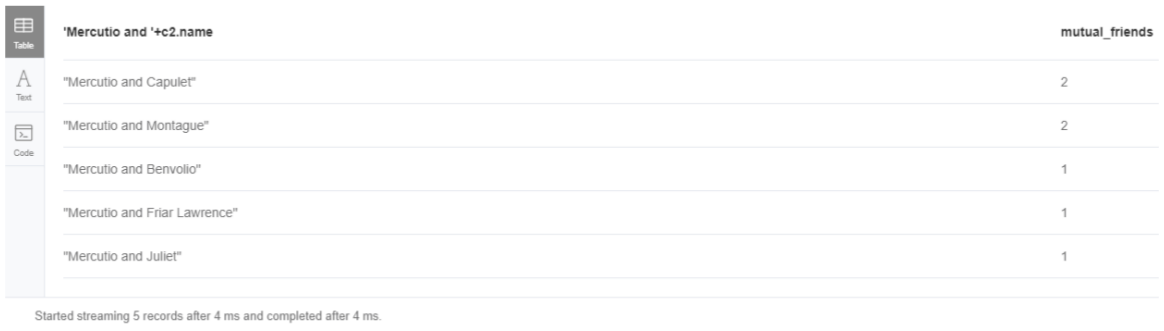
Similarly, count the number of mutual friends between Mercutio and Capulet. Make sure the names Mercutio and Capulet are followed by the number of mutual friends between them. Your results should look like the following picture.



'Mercutio and Capulet'	mutual_friends
"Mercutio and Capulet"	2

Started streaming 1 records after 4 ms and completed after 4 ms.

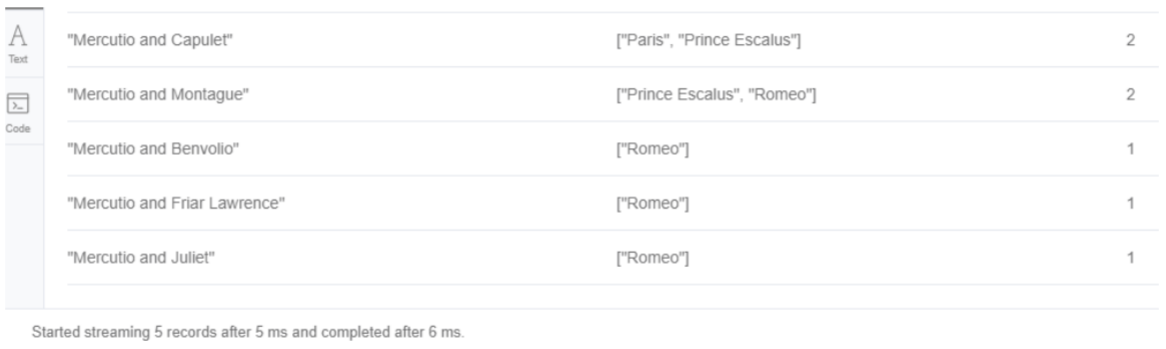
Now, you should produce a generic CQL query to list the number of mutual friends between Mercutio and each character of Romeo and Juliet who is not yet a friend of Mercutios. It is not necessary to list the names of characters who do not have any mutual friends with Mercutio. However, make sure you sort the list by the number of mutual friends in *descending* order - the characters with the largest number of mutual friends should appear at the top. The results should look like the following picture.



'Mercutio and '+c2.name	mutual_friends
"Mercutio and Capulet"	2
"Mercutio and Montague"	2
"Mercutio and Benvolio"	1
"Mercutio and Friar Lawrence"	1
"Mercutio and Juliet"	1

```
Started streaming 5 records after 4 ms and completed after 4 ms.
```

Produce a query to obtain not only the number of mutual friends between Mercutio and the rest of the characters, but also the names of the mutual friends. Once again, it is not necessary to list the names of characters who do not have any mutual friends with Mercutio. However, make sure you sort the list by the number of mutual friends in *descending* order (the characters with the largest number of mutual friends should appear at the top). Your results should look like the following picture.



"Mercutio and Capulet"	["Paris", "Prince Escalus"]	2
"Mercutio and Montague"	["Prince Escalus", "Romeo"]	2
"Mercutio and Benvolio"	["Romeo"]	1
"Mercutio and Friar Lawrence"	["Romeo"]	1
"Mercutio and Juliet"	["Romeo"]	1

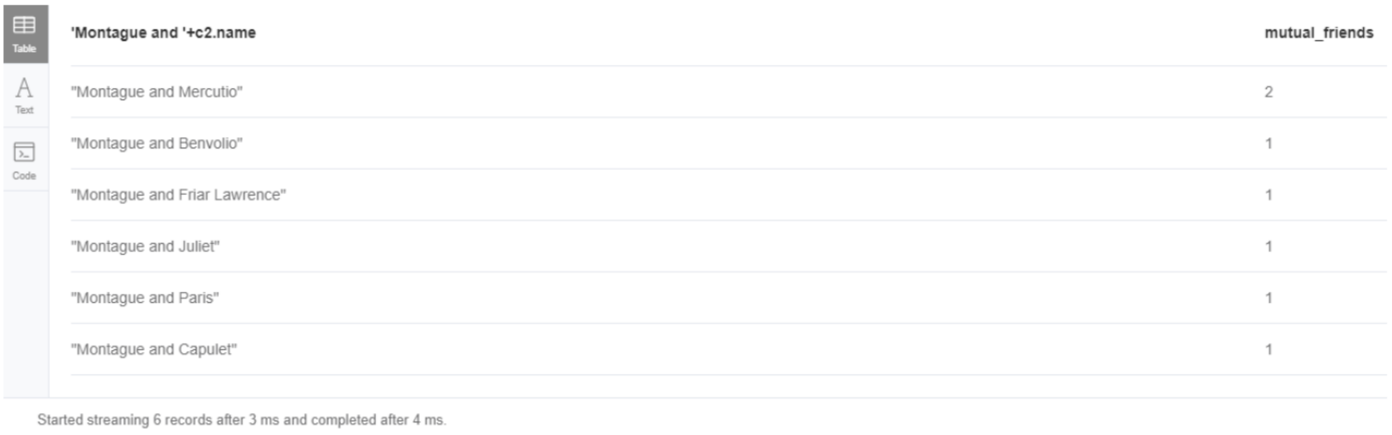
```
Started streaming 5 records after 5 ms and completed after 6 ms.
```

According to the collaborative filtering algorithm, the best friend recommendation for Mercutio is the character with whom Mercutio has the largest number of mutual friends. In our case, the results you have just obtained show that:

- Mercutio has two friends in common with Capulet (Paris and Prince Escalus).
- Mercutio has two friends in common with Montague (Prince Escalus and Romeo).
- Mercutio has one friend in common with Benvolio (Romeo).
- Mercutio has one friend in common with Friar Lawrence (Romeo).
- Mercutio has one friend in common with Juliet (Romeo).

Therefore, Capulet and Montague are the best friend recommendations for Mercutio.

Following the same approach used above, produce a generic query to list the number of mutual friends between Montague and each character of Romeo and Juliet who is not yet a friend of Montagues. Once again, it is not necessary to list the names of characters who do not have any mutual friends with Montague. However, make sure you sort the list by the number of mutual friends in *descending* order (the characters with the largest number of mutual friends should appear at the top). The results should look like the following picture.



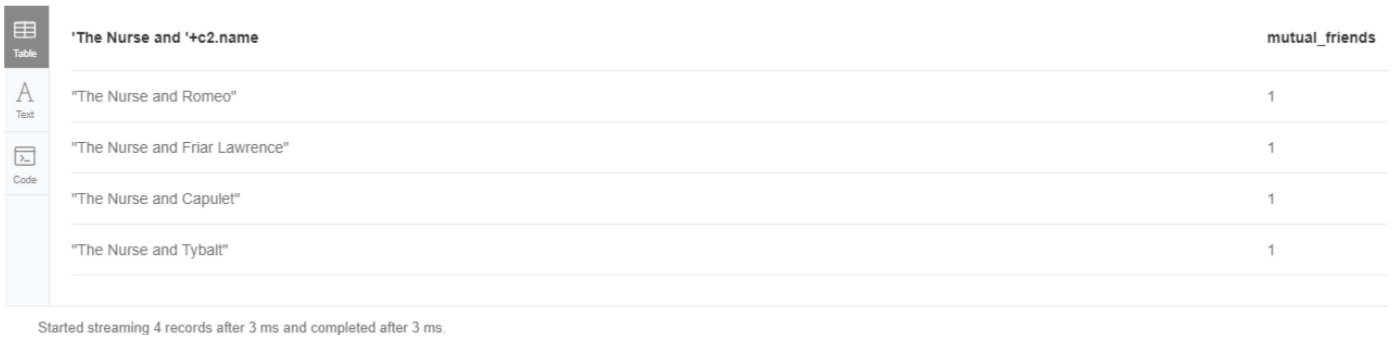
'Montague and '+c2.name	mutual_friends
"Montague and Mercutio"	2
"Montague and Benvolio"	1
"Montague and Friar Lawrence"	1
"Montague and Juliet"	1
"Montague and Paris"	1
"Montague and Capulet"	1

Started streaming 6 records after 3 ms and completed after 4 ms.

Therefore, according to the collaborative filtering algorithm, Mercutio is the best friend recommendation for Montague.

To finish this practical, identify the best friend recommendation for each of the characters in Romeo and Julietthis is the best friend recommendation for Romeo, Benvolio, Prince Escalus, Paris, Juliet, Capulet, Friar Lawrence, The Nurse and Tybalt.

While the collaborative filtering algorithm is relatively easy to implement, it has a number of limitations. For example, the Nurse has only one friend in common with four different characters, and it is impossible to distinguish which one of them constitutes the best friend recommendation.



'The Nurse and '+c2.name	mutual_friends
"The Nurse and Romeo"	1
"The Nurse and Friar Lawrence"	1
"The Nurse and Capulet"	1
"The Nurse and Tybalt"	1

Started streaming 4 records after 3 ms and completed after 3 ms.