CSCI 5751 Spring 2021 Project 1 Phase 3

Group Name: Avatar

Video Link: [Phase 3]

Xiang Zhang
University of Minnesota, Twin Cities
zhan6668@umn.edu

Zixuan Zhang
University of Minnesota, Twin Cities
zhan7230@umn.edu

Wei Ye
University of Minnesota, Twin Cities
ye000094@umn.edu

Luyao Zhang
University of Minnesota, Twin Cities
zhan5380@umn.edu

Outline

0. Introduction

Our business questions are about developing a useful movie recommender system for customers and film studios. Since we need to create a database with flexibility in expanding the model to add new entities or data relationships and convenience to add or remove relationships, we chose the graph database. Among many graph databases, we found that Neo4j can store information in the form of an edge, node, or attribute. Each node and edge can have any number of attributes. Both nodes and edges can be labelled, and labels can be used to narrow searches. All these properties met our requirements, so we picked Neo4j as the main tool. With the graph database, we can do relational queries really fast by omitting joining. For example, we can get which actor once cooperated with a specific director in a very short time with a simple query based on the relationships saved as links in our graph data models. We stored the relationships (rate/acted_in/directed) between nodes for different individuals (customers/cast/directors) and nodes for movies as shown in Figure 1, the visualization of the nodes and links of an instance queried in our Neo4j database.

In our project, we implemented this database at different cloud technology levels. We firstly constructed this graph database on a remote virtual machine (Infrastructure as a Service) for good and stable performance. Then, we also realized it on AWS (Software as a Service). For our ultimate business question, we managed to provide 3 kinds of different recommendations, including recommendation based on the movie genres, recommendation based on the similar customers, and recommendation based on customers' favorite cast.

By doing this proof of concept successfully, we have learned that: 1) proper data model design and data manipulation boosted the efficiency of solving questions with faster queries; 2) cloud deployment of NoSQL database allows more flexible management and scalable performance; 3) we gradually adapted to the new mindset based on graph models when solving relationship-based questions.

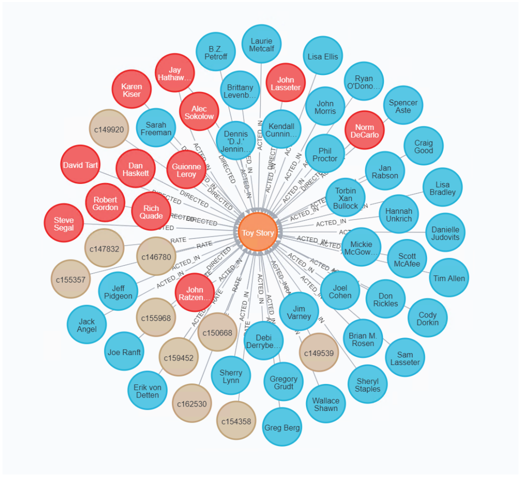


Figure 1. Visualization of an example from our graph database. We designed four kinds of node labels (movies, actors/actresses, directors, customers) and three kinds of relationships (ACTED_IN, DIRECTED, RATE)

Key links:
Raw data; Manipulated data; Github repository;

1. Data used for analysis

Our datasets cover two sources: (1) MovieLens dataset in CSV format from GroupLens; (2) IMDb dataset in TSV format. We set MovieLens as our key dataset and retrieved other information from IMDb such as cast members, directors, and average ratings on IMDb. Specifically, in Figure 2 below, we illustrate the content of datasets and their inherent relationships:

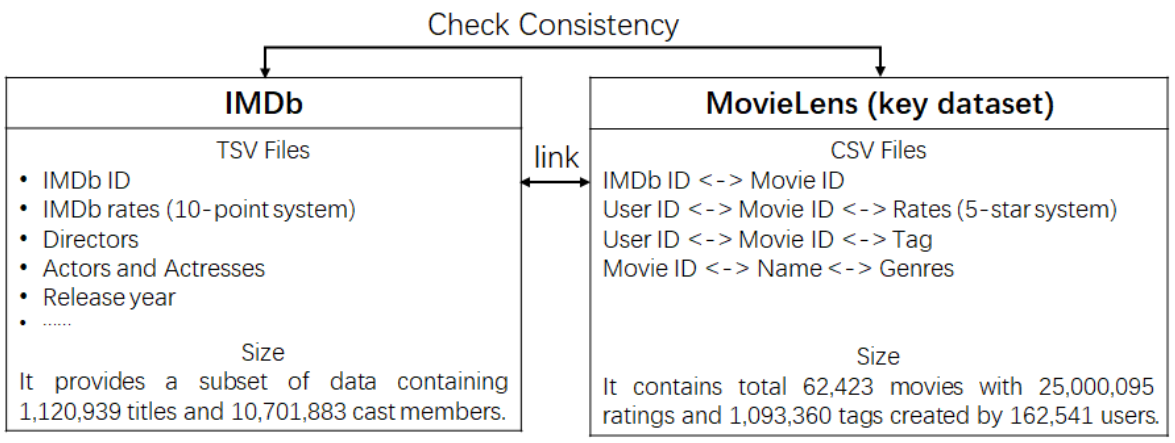

Check Consistency
IMDb
TSV Files
• IMDb ID
• IMDb rates (10-point system)
• Directors
• Actors and Actresses
• Release year
• ......
Size
It provides a subset of data containing 1,120,939 titles and 10,701,883 cast members.
link
MovieLens (key dataset)
CSV Files
IMDb ID <-> Movie ID
User ID <-> Movie ID <-> Rates (5-star system)
User ID <-> Movie ID <-> Tag
Movie ID <-> Name <-> Genres
Size
It contains total 62,423 movies with 25,000,095 ratings and 1,093,360 tags created by 162,541 users.

Figure 2. Overview of our selected raw datasets from IMDb and MovieLens.

MovieLens is a movie recommendation service run by GroupLens, a research lab at the University of Minnesota. This data was created between January 09, 1995 and November 21, 2019.

Internet Movie Database (IMDb) is the world's most popular and authoritative source for movie, TV, and celebrity content. The IMDb official website provides a subset of data containing a great amount of information about movies and relevant features. We retrieved this daily refreshed data on February 7th, 2021.

In our data model, there will be four kinds of node labels and three kinds of relationships as shown in Figure 3. Our recommendation system uses two aspects of information: attribute and structure. These two indicators can be easily implemented in the graph database, while in other systems it costs a lot. More details about our data model can be found in Appendix B and Phase 2 report. Note that after data pre-processing (see Section 3 for details), the scale of data got changed, and thus some numbers in Figure 3 may differ from those in Figure 2.

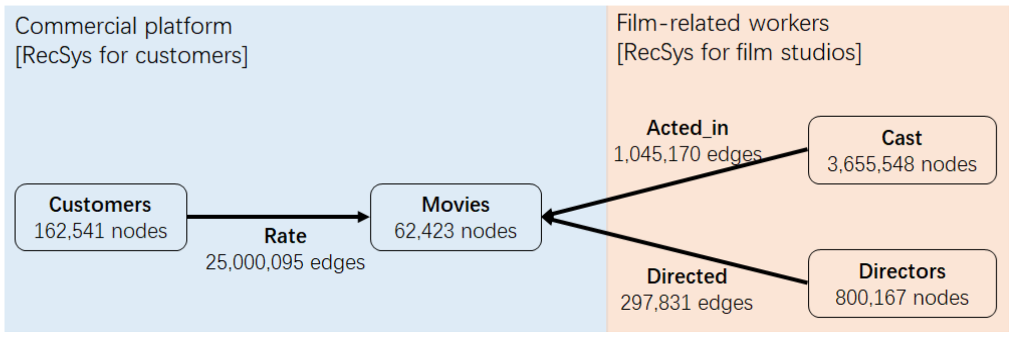


Figure 3. Overview of our data model design.

2. The storage technology we used (Neo4j) and its main features

Why NoSQL graph databases?

(1) We have multiple data sources in large size, each of them with a unique structure. Thus, we need a database with good flexibility, and we need to model real-world information comprising high fidelity and complexity. Those requirements ruled out most simple RDBMS (Relational Database Management System) and made NoSQL a good candidate.

(2) However, we do care about the valuable relationships within the movie data. Moreover, our datasets are not limited to just one-to-many relationships, but are highly connected by many-to-many relationships. For example, when we try to investigate the relationship between a group of directors and movies they directed, there will be multiple directors related to multiple movies. Therefore, graph databases became a preferable choice compared to other NoSQL technologies such as document storage.

(3) Our project is an OLAP system, which applies complex queries to large amounts of data, and emphases the response time to these complex queries. Since each query involves one or more columns of data aggregated from many rows, we need a database that can save and manage the relationships directly.

Why Neo4j as our NoSQL technology?

Some graph databases provide a declarative query language while others provide only imperative ways. As application programmers, we want to keep our application code as clean as possible without worrying about the implementation details and query optimizations of our databases. Therefore, among several graph databases, we chose Neo4j for its declarative query language --- Cypher. In addition, the Neo4j community provided well-written documents and tutorials that supported us well.

3. Manipulations required to use our data (with code)

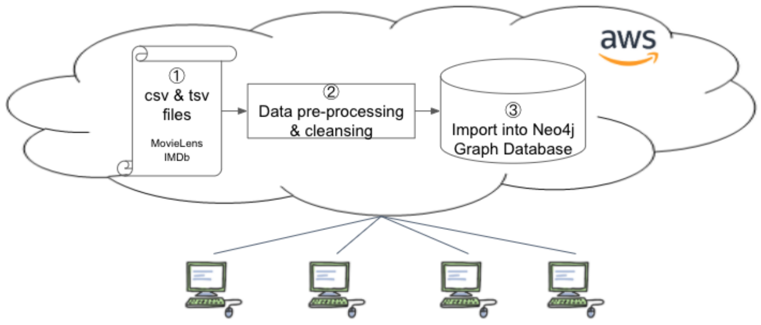

aws
①
csv & tsv files
MovieLens
IMDb
②
Data pre-processing & cleansing
③
Import into Neo4j Graph Database

Figure 4. Dataflow for ingesting our data into the system.

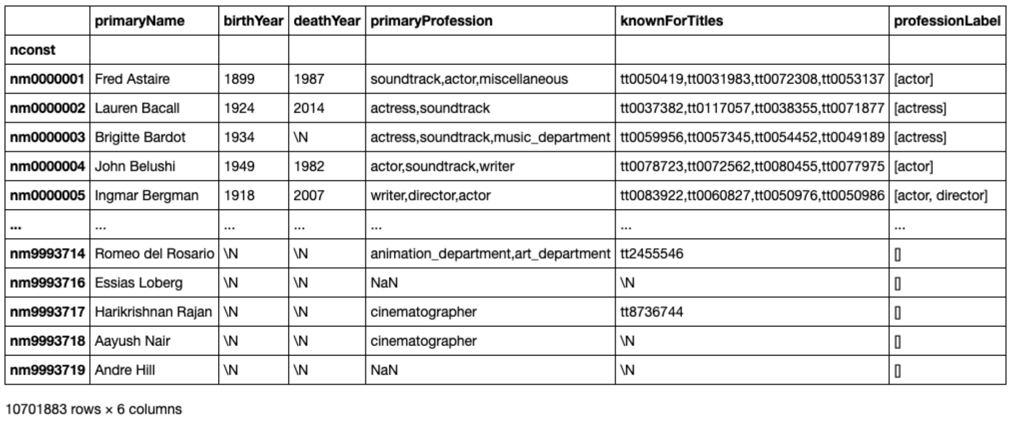
(1) Download:

We acquired raw data files from MovieLens and IMDb as specified in Appendix A, which provide the information that can be joined together for our nodes and links.

(2) Pre-processing:

We stored the whole data-preprocessing pipeline in our Github repository. Some manipulations to highlight:

i) We used Python to pre-process and cleanse the data in a notebook that visualizes our analysis. While most data entries in the CSV files we acquired from MovieLens and IMDb are valid and conform to the same format, some entries inevitably contain empty or invalid fields. For example, in the dataset *name.basics.tsv* recording the information of 10,701,883 individuals as shown in Figure 5, there are 2,246,089 rows that do not record any profession information for the person, which would not be helpful for us to construct relationships between these people and the movies. So we excluded these nodes before ingesting the data into the graph database.

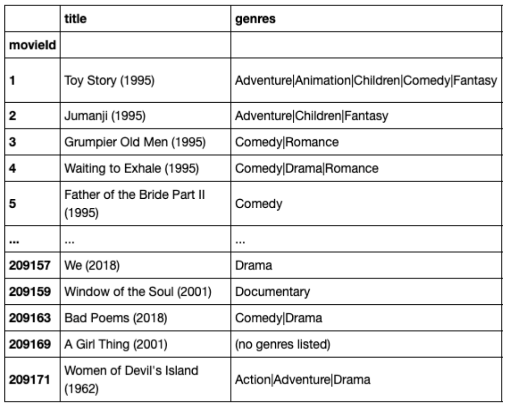


	primaryName	birthYear	deathYear	primaryProfession	knownForTitles	professionLabel
nconst						
nm0000001	Fred Astaire	1899	1987	soundtrack,actor,miscellaneous	tt0050419,tt0031983,tt0072308,tt0053137	[actor]
nm0000002	Lauren Bacall	1924	2014	actress,soundtrack	tt0037382,tt0117057,tt0038355,tt0071877	[actress]
nm0000003	Brigitte Bardot	1934	\N	actress,soundtrack,music_department	tt0059956,tt0057345,tt0054452,tt0049189	[actress]
nm0000004	John Belushi	1949	1982	actor,soundtrack,writer	tt0078723,tt0072562,tt0080455,tt0077975	[actor]
nm0000005	Ingmar Bergman	1918	2007	writer,director,actor	tt0083922,tt0060827,tt0050976,tt0050986	[actor, director]
...	...	...	...	...	...	...
nm9993714	Romeo del Rosario	\N	\N	animation_department,art_department	tt2455546	[]
nm9993716	Essias Loberg	\N	\N	NaN	\N	[]
nm9993717	Harikrishnan Rajan	\N	\N	cinematographer	tt8736744	[]
nm9993718	Aayush Nair	\N	\N	cinematographer	\N	[]
nm9993719	Andre Hill	\N	\N	NaN	\N	[]

10701883 rows × 6 columns

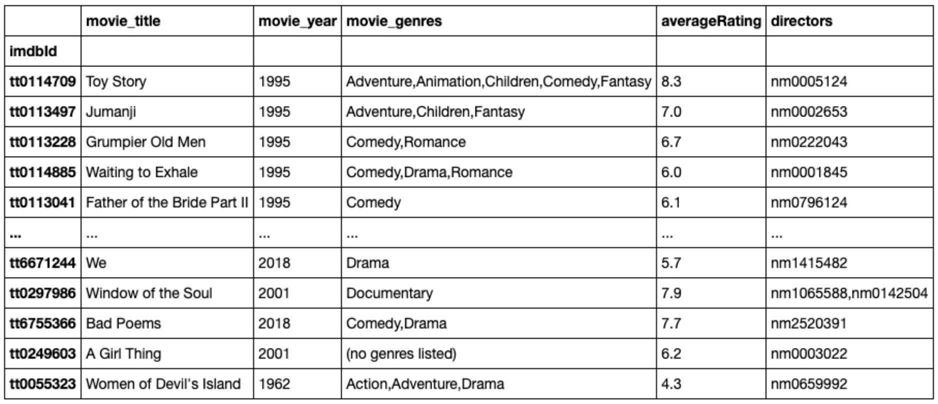
Figure 5. Snapshot of the file *name.basics.tsv* recording 10,701,883 individuals.

ii) We formatted some attributes whose original annotations would make it hard to process directly by the Cypher query language. However, for a Python script, especially using Pandas, this is not a big deal. Take the file *movies.csv* for instance. Instead of having separate columns for title and release year, it stores them together with the format of 'Movie Name (Year)'. While it is possible to extract these two values using Cypher, the extraction would be much easier to perform using Python. As shown in Figure 6 below, after reformatting the titles and genres, as well as merging the ratings from IMDb, this MovieLens data evolved from (a) to (b) so that we can solve our business questions more efficiently with clearer Cypher queries.



	title	genres
movieId		
1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
2	Jumanji (1995)	Adventure|Children|Fantasy
3	Grumpier Old Men (1995)	Comedy|Romance
4	Waiting to Exhale (1995)	Comedy|Drama|Romance
5	Father of the Bride Part II (1995)	Comedy
...	...	...
209157	We (2018)	Drama
209159	Window of the Soul (2001)	Documentary
209163	Bad Poems (2018)	Comedy|Drama
209169	A Girl Thing (2001)	(no genres listed)
209171	Women of Devil's Island (1962)	Action|Adventure|Drama

(a) Snapshot of original data



	movie_title	movie_year	movie_genres	averageRating	directors
imdbId					
tt0114709	Toy Story	1995	Adventure,Animation,Children,Comedy,Fantasy	8.3	nm0005124
tt0113497	Jumanji	1995	Adventure,Children,Fantasy	7.0	nm0002653
tt0113228	Grumpier Old Men	1995	Comedy,Romance	6.7	nm0222043
tt0114885	Waiting to Exhale	1995	Comedy,Drama,Romance	6.0	nm0001845
tt0113041	Father of the Bride Part II	1995	Comedy	6.1	nm0796124
...	...	...	...	...	...
tt6671244	We	2018	Drama	5.7	nm1415482
tt0297986	Window of the Soul	2001	Documentary	7.9	nm1065588,nm0142504
tt6755366	Bad Poems	2018	Comedy,Drama	7.7	nm2520391
tt0249603	A Girl Thing	2001	(no genres listed)	6.2	nm0003022
tt0055323	Women of Devil's Island	1962	Action,Adventure,Drama	4.3	nm0659992

(b) Snapshot of manipulated data

Figure 6. The evolution of the table where we manage the movie data.

iii) Another important step during data pre-processing is that we need to join some informative columns to the key column from different tables in order to facilitate the process when importing data into our Neo4j database. Therefore, we re-assigned uniform movie IDs and customer IDs in order to yield consistent keys. This was also not a hard job for Python. The design of our nodes and links is a good reference for the final tables after joining.

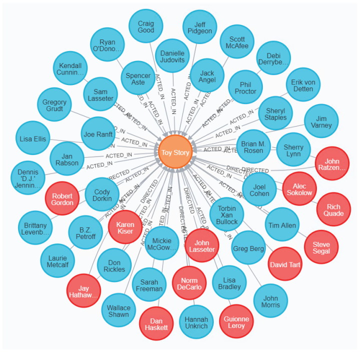
Eventually, we utilized Python to produce cleansed input files that are all in CSV format to be ingested into the Neo4j graph database, which are stored in a shared google drive folder input_Neo4j.

(3) Import:
With the pre-processing and cleansing done, as well as essential columns joined together for each node by Python, we created a project and a Neo4j instance on a remote virtual machine which provided us with a larger memory than our own laptops and the convenience for cooperation. Then, we used Cypher to load the selected CSV files, and finally constructed the nodes and relationships strictly following our data model. The interaction with this graph database was demonstrated in our video for Phase 3.

4. Answers to our analytical questions (with code)

Table 1 presents the answers to all our business questions. We also further describe some of the questions and our strategies. In Appendix D we show the screenshots of running the query code.

Table 1. Questions about movie rating start with Index A. Questions about actor/director start with Index B. Questions about recommending the movie to customers start with Index C.

Easy	**A1. How many movies are there in the database?**
Code	`MATCH (m:Movie) RETURN COUNT(m)`
Result	Count = 62423
Easy	**A2. How many null values are there in the *primaryProfession* column in the cast name dataset?**
Description	This question was solved during data-preprocessing using Python. The complete context and visualized analysis can be found in the Github link provided in Section 3.2.
Code	`print(df_name[df_name['primaryProfession'].isnull()].shape[0])`
Result	2,246,089
Easy	**A3: Who are the directors and casts known for the movie "Toy Story"?**
Code	`MATCH p=(a:Actor)-[ACTED_IN]->(m:Movie {title: 'Toy Story'})<-[DIRECTED]-(d:Director) RETURN p`
Result	

Easy	**A4. What is the max/min/avg of the MovieLens ratings for "Avatar"?**
Code	`MATCH ()-[r:RATE]->(m:Movie {title: 'Avatar'}) RETURN max(toInteger(r.rating)), min(toInteger(r.rating)), avg(toInteger(r.rating))`

Result

Max rating: 5
Min rating: 0
Avg rating: 3.423578

Moderate — **B1. How many movies Christian Bale and Michael Caine both acted in?**

Code

```
MATCH (a1:Actor {name: 'Christian Bale'})-[:ACTED_IN]->(m:Movie)<-[:ACTED_IN]-(b:Actor
{name: 'Michael Caine'})
RETURN count(m)
```

Result

Count = 1

Moderate — **B2. How many directors Christian Bale and Michael Caine worked with?**

Code

```
MATCH (a1:Actor {name: 'Christian
Bale'})-[:ACTED_IN]->(m1:Movie)<-[:DIRECTED]-(d:Director)-[:DIRECTED]->(m2:Movie)<-[ACTED
_IN]-(a2:Actor {name: 'Michael Caine'})
RETURN count(d)
```

Result

Count = 7

Moderate — **C1. Which actor/actresses does a specific customer probably like the most, who we can recommend to this customer in the future, based on all the movies he/she rated?**

Code

```
MATCH (c:Customer {id: 'c000001'})-[:RATE]->(m:Movie)<-[:ACTED_IN]-(a:Actor)
RETURN a, count(a)
ORDER BY count(a) DESC
LIMIT 1
```

Result

- The person we find with "Count = 4":

```
{"identity": 324676,
"labels": ["Actor"],
"properties": {"name": "Borje Lundh", "id": "nm0526464"}}
```

Challenging — **C2. Which movie should be recommended to a specific customer?**

Description

In this question, we hope to achieve personalized recommendation of movies to customers. By observing the existing commercial recommendation systems, we found that their recommendation strategies often consider various factors. Specifically, they will consider the genres of movies users like, the behaviors of similar users, and favorite actors or directors, etc. Therefore, in this project, we also initially realized these functions.

Method 1

Recommendation based on the movie genres: Each customer has different preferences for the genres of movie. For example, some customers are very enthusiastic about science fiction but may dislike horror movies. At this time, the number of sci-fi movies rated by customers will be much greater than that of horror movies. Therefore, we can make recommendations based on the watching preferences of this customer.

- Specifically, there will be two steps:

Step 1. Find the genre of the most frequently watched the movie by this customer;
Step 2. Find the movie in this genre with the highest IMDb rating that he/she hasn't watched.

- Code for step 1:

```
MATCH (c1:Customer {id: 'c072345'})-[:RATE]->(m:Movie)
UNWIND m.genres AS genre
RETURN genre, COUNT(genre)
ORDER BY COUNT(genre) DESC
LIMIT 1
```

- Code for step 2:

```
MATCH (c:Customer {id: 'c072345'})-[:RATE]->(m1:Movie) WITH collect(m1) AS movies
MATCH   (m2:Movie)   WHERE   'Comedy'   IN   m2.genres   AND   NOT(m2   in   movies)   AND
EXISTS(m2.IMDb_rating)
RETURN m2, m2.IMDb_rating
ORDER BY m2.IMDb_rating DESC
LIMIT 1
```

- Results:

Result for step 1:
Genre: “Drama” Count: 53

Result for step 2:
{"identity": 60816,
"labels": ["Movie"],
"properties": {"id": "tt6643114", "title": "07/27/1978", "year": 2017, "genres": ["Comedy", "Documentary"], "IMDb_rating": 9.6}}

Method 2

Recommendation based on the similar customer: Although the watching preferences between customers may not be exactly the same, there is indeed a certain degree of similarity between among them, which can be used as a good indicator for recommendation. This is often an efficient method especially in the context of the big data era.

- Specifically, there will be two steps:

Step 1. Find the most similar customer to our interested customer based on watching history;
Step 2. Recommend the movie with the IMDb highest rating from that similar person.

- Code for step 1:

```
MATCH (c1:Customer {id: 'c000001'})-[:RATE]->(m:Movie)<-[:RATE]-(c2:Customer)
RETURN c2, COUNT(c2)
ORDER BY COUNT(c2) DESC
LIMIT 1
```

- Code for step 2:

```
MATCH (c1:Customer {id: 'c000001'})-[:RATE]->(m1:Movie) WITH collect(m1) AS movies
MATCH (c2:Customer {id: 'c072315'})-[r:RATE]->(m2:Movie)
WHERE NOT(m2 IN movies)
RETURN m2, r.rating
ORDER BY r.rating DESC
LIMIT 1
```

- Results:

Result for step 1:
{"identity": 4422635,
"labels": ["Customer"],
"properties": {"id": "c072315"}}

Result for step2:
{"identity": 9760,
"labels": ["Movie"],
"properties": {"id": "tt0026955", "title": "Ruggles of Red Gap", "year": 1935, "genres": ["Comedy", "Romance"], "IMDb_rating": 7.6}}

Method 3

Recommendation based on the favorite actor/actress: Famous actors or actresses often have a large group of fans who will enthusiastically support their idol. For example, they will watch all the works they have filmed or starred in. Whether the system can detect their favorite stars and recommend corresponding works can greatly affect the user experience of these customers.

- Specifically, there will be two steps:

Step1. Find a customer’s favorite actor/actress;
Step2. Find the highest-rated IMDb movie that this actor/actress acted in.

- Code for step 1:

```
MATCH (c:Customer {id: 'c033301'})-[:RATE]->(m:Movie)<-[:ACTED_IN]-(a:Actor)
RETURN a, COUNT(a)
ORDER BY COUNT(a) DESC
LIMIT 1
```

- Code for step 2:

```
MATCH (c:Customer {id: 'c033301'})-[:RATE]->(m1:Movie) WITH collect(m1) AS movies
```

	`MATCH (m2:Movie)<-[:ACTED_IN]-(a:Actor {name: "Grover Richardson"})`
	`WHERE NOT(m2 IN movies)`
	`RETURN m2, m2.IMDb_rating`
	`ORDER BY m2.IMDb_rating DESC`
	`LIMIT 1`
	● Results:	

	Result for step 1: {"identity": 1376470, "labels": ["Actor"], "properties": {"name": "Grover Richardson", "id": "nm2301534"}}	Result for step 2: {"identity": 11698, "labels": ["Movie"], "properties": {"id": "tt0418279", "title": "Transformers", "year": 2007, "genres": ["Action", "Sci-Fi", "Thriller", "IMAX"], "IMDb_rating": 7.0}}

5. Lessons learned about NoSQL using public movie data and Neo4j

Lessons learned from using the NoSQL technology:

Choosing the right tool is very important. Different NoSQL has its own advantages and disadvantages. For example, we want to build a recommendation system that focuses on the relationship between instances and themselves. Therefore, a graph database is a good choice. But the graph database is not suitable for storing large binary objects. For example, we originally wanted to display the recommendation results in the form of movie posters, but if the movie poster was stored as the attribute of the movie node, then database would go down (a decrease of performance https://neo4j.com/blog/dark-side-neo4j-worst-practices/). We believe that the solution to this problem is to first obtain the id of a recommended movie through the graph database, and then retrieve the poster from another database which can be built upon other NoSQL technologies such as key-value storage.

Lessons learned from using the public movie data:

(1) A user manual with clear and detailed descriptions is very important. When we need to use public data sets, we should first evaluate these documents, and see what kind of business problems these data can help solve. Then, we can design data models and download the required parts. Because of inexperience, we downloaded a lot of data at the beginning of the project, while a great proportion of them were not used in the end. We were swamped by the large number of files at the very beginning which made it harder to raise clear questions.

(2) Multiple data sets are needed. Each single data source may only include a limited range of data and thus cannot meet all our needs. In our project, we used two data sources, MovieLens and IMDb, to solve our business questions which required fields from both datasets. However, when using multiple data sources, it is very important to check the consistency of the data. For example, a movie may have different names even in different languages recorded in different databases, so it is significant to align them with a uniform id.

(3) Data pre-processing is necessary and time-consuming. The public dataset is often not very clean, and the format of some attributes in the dataset may not be what we expect. For example, movie ratings may be stored as a sting type instead of a float type, adding difficulties in doing mathematical operations on it. This requires us

to do data pre-processing according to our designed data model and needs for analysis. Because we did not check all the data types read in the database at initial trials, such issues had us spend more time back and forth.

Lessons learned from using the Neo4j graph database:
A new mindset based on graph databases has been adapted. Unlike other NoSQL databases, ours emphasizes the perspective of relationships. From a high-level point of view, solving a problem using the traditional database management systems is more like based on set theory (intersection, union), while for using graph databases it is more like based on graph theory (connection structure, information dissemination). Therefore, the way of thinking in programming is distinct from the traditions. Through this project, we not only learned a new tool, but also a new mindset. Table 2 below presents a comparison between the way we think in the operation of "intersection" with SQL based on Set theory and with Cypher based on Graph theory.

Table 2. Different styles of query code demonstrating different mindsets

Intersection in Set theory	Intersection in Graph theory
SELECT column1 FROM table1 INTERSECT SELECT column1 FROM table1	n1:Node)-[r1:relationship]->(m:CommonNode)<-[r2 ship]-(n2:Node)

6. Challenges we faced

(1) Encountering large files that require partition:

When we were loading our cleansed data into the Neo4j, we encountered an "out of memory" issue when loading the file recording ~25 million ratings to create nodes for 162,541 customers. We concluded that this should be caused by the attempt to load the whole dataset into the memory by the system. Accordingly, we partitioned this large file into 10 pieces and read them into the database successfully, without having to remove any valuable information that could be useful for our recommender system.

(2) The datasets of cast and directors do not record all the movies they were involved in:
The source file from IMDb recording the names of individuals (*name.basics.tsv*) includes the movie titles that these actors/actresses/directors were known for, but the number for each individual's work was limited to 4, which means it is definitely incomplete compared to the real world case, i.e. an actor/actress can have experience in acting in more than 4 movies. This would limit the power of our recommender system if we adopt the Method 3 described in the business question **C2** above - recommend a movie to users that their favorite actors/actresses acted in but they have not watched yet.

(3) Implementation of advanced similarity or link prediction algorithms:
When we wanted to further implement more complicated similarity algorithms for our movie recommender system based on the similarity among customers, we found that the implementations require a deeper grasp of the Cypher language as well as the mechanism of Neo4j. Moreover, for some algorithms about common neighbors, it is necessary to transfer the current bipartite graphs between nodes for customers and nodes for

movies into a monopartite graph only including customers. Since we did not have sufficient time to dive in these advanced algorithms, we developed three methods from different perspectives to recommend movies to a customer as described in business question **C2** above, which is also showing great power in recommendation.

7. Proof of concept - what worked and did not work

(1) We proved that it is feasible to integrate the public movie datasets into a graph representation in a database with some manipulations. The relationships are presented by the form of nodes and links, which enables a highly efficient strategy in exploring the relationships throughout the database on which graph algorithms can be applied.

(2) The design of the data model matters a lot. There could be various ways of constructing nodes and relationships. For example, "actor/actress/director" can be labels belonging to the same type of node, or they can just be different types of nodes respectively. Our final design was intended to best fit our data as well as business questions.

(3) Installing our main technology (Neo4j) on either local desktop or remote cloud services worked. We successfully deployed our database onto the cloud at the level of IaaS (virtual machine) and SaaS (AWS). However, without the development of front end, we were not able to establish a complete commercial pipeline.

(4) After ingesting our manipulated data into the Neo4j database, we also did experiments about dealing with the null values in some other fields that were kept, which worked well without very complicated Cypher queries. But this is the case where we only want to analyze the data instead of manipulating the data from structure. Otherwise we would have to keep track of manipulations and be aware of the dynamically changed system.

(5) Currently, we can still use Python to process these files, each with a size of megabyte-level. However, this could not guarantee the scalability of our pre-processing pipeline. We may not be able to use Python for pre-processing if the incoming data file is too large to do any manipulation with. Accordingly, we thought of Spark and were interested in giving it a try, as recently there has been a Neo4j Connector for Apache Spark attracting attention. However, as there are many as well as immature prerequisites for building and integrating Neo4j with Apache Spark applications, we did not make it work. But this is definitely a worthwhile attempt and possibly the future direction in this field.

8. Appendix

Appendix A
Description of the raw datasets

We backup our raw data in a google drive folder: Data to use

Description of downloaded data:
1. MovieLens 25M Dataset: **ml-25m.zip** (250 MB, 1.16Gb after being unzipped)

This dataset (ml-25m) describes 5-star rating and free-text tagging activity from MovieLens, a movie recommendation service. It contains 25,000,095 ratings and 1,093,360 tag applications across 62,423 movies. These data were created by 162,541 users between January 09, 1995 and November 21, 2019. This dataset was generated on November 21, 2019.

Users were selected at random for inclusion. All selected users had rated at least 20 movies. No demographic information is included. Each user is represented by an id, and no other information is provided.

The data are contained in the files **genome-scores.csv**, **genome-tags.csv**, **links.csv**, **movies.csv**, **ratings.csv** and **tags.csv**. More details about the contents and use of all these files are in the Readme file.

2. IMDb Dataset:
title.akas.tsv.gz (207 MB) - Contains the following information for titles:
- titleId (string) - a tconst, an alphanumeric unique identifier of the title
- ordering (integer) – a number to uniquely identify rows for a given titleId
- title (string) – the localized title
- region (string) - the region for this version of the title
- language (string) - the language of the title
- types (array) - Enumerated set of attributes for this alternative title. One or more of the following: "alternative", "dvd", "festival", "tv", "video", "working", "original", "imdbDisplay".
- attributes (array) - Additional terms to describe this alternative title, not enumerated
- isOriginalTitle (boolean) – 0: not original title; 1: original title

title.basics.tsv.gz (129 MB) - Contains the following information for titles:
- tconst (string) - alphanumeric unique identifier of the title
- titleType (string) – the type/format of the title (e.g. movie, short, tvseries, tvepisode, video, etc)
- primaryTitle (string) – the more popular title / the title used by the filmmakers on promotional materials at the point of release
- originalTitle (string) - original title, in the original language
- isAdult (boolean) - 0: non-adult title; 1: adult title

- startYear (YYYY) – represents the release year of a title. In the case of TV Series, it is the series start year
- endYear (YYYY) – TV Series end year. ‘\N’ for all other title types
- runtimeMinutes – primary runtime of the title, in minutes
- genres (string array) – includes up to three genres associated with the title

title.crew.tsv.gz (50 MB) – Contains the director and writer information for all the titles in IMDb. Fields include:
- tconst (string) - alphanumeric unique identifier of the title
- directors (array of nconsts) - director(s) of the given title
- writers (array of nconsts) – writer(s) of the given title

title.episode.tsv.gz (29 MB) – Contains the tv episode information. Fields include:
- tconst (string) - alphanumeric identifier of episode
- parentTconst (string) - alphanumeric identifier of the parent TV Series
- seasonNumber (integer) – season number the episode belongs to
- episodeNumber (integer) – episode number of the tconst in the TV series

title.principals.tsv.gz (334 MB) – Contains the principal cast/crew for titles
- tconst (string) - alphanumeric unique identifier of the title
- ordering (integer) – a number to uniquely identify rows for a given titleId
- nconst (string) - alphanumeric unique identifier of the name/person
- category (string) - the category of job that person was in
- job (string) - the specific job title if applicable, else '\N'
- characters (string) - the name of the character played if applicable, else '\N'

title.ratings.tsv.gz (5 MB) – Contains the IMDb rating and votes information for titles
- tconst (string) - alphanumeric unique identifier of the title
- averageRating – weighted average of all the individual user ratings
- numVotes - number of votes the title has received

name.basics.tsv.gz (129 MB) – Contains the following information for names:
- nconst (string) - alphanumeric unique identifier of the name/person
- primaryName (string)– name by which the person is most often credited
- birthYear – in YYYY format
- deathYear – in YYYY format if applicable, else '\N'
- primaryProfession (array of strings)– the top-3 professions of the person
- knownForTitles (array of tconsts) – titles the person is known for

Data files we eventually used:

From GroupLens-MovieLens-25m:
1. links.csv: movieId, imdbId, tmdbId
2. movies.csv: movieId, title, genres
3. ratings.csv: userId, movieId, rating, timestamp

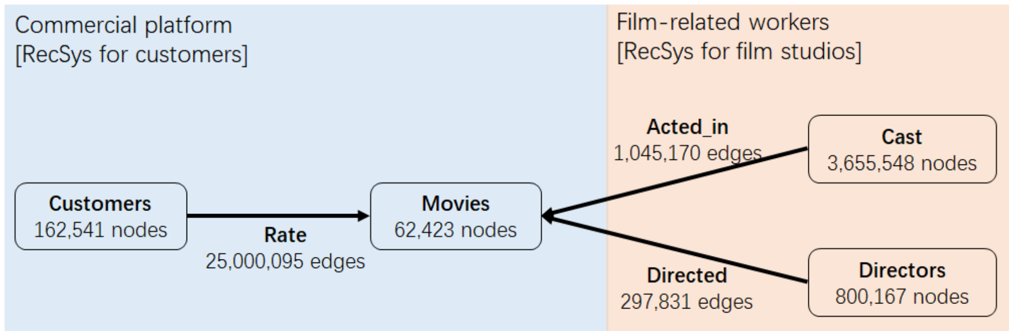
From IMDb_official_dataset_20210207:
1. name.basics.tsv
2. title.ratings.tsv
3. title.akas.tsv
4. title.crew.tsv

We store the manipulated data, which is our final input to the Neo4j database, in another shared google drive folder input_Neo4j.

Appendix B
Data model design

B.1 Brief description about the data model:

Our business questions are about developing recommender systems for customers and film studios. In our data model, there will be four kinds of node labels and three kinds of relationships as shown in the following figure.



B.2 High-level motivation for model design:

Since we are mainly interested in the relationships among entities, on which we can do querying as well as operations quickly within a large scale of data, we chose to build a graph database system. Our recommendation system uses two aspects of information: attribute and structure. From the perspective of properties, graph databases are easy to perform relationship-based queries with. From a structural point of view, we need to use many graph properties such as various node centrality indicators. These centrality indicators can be easily implemented on the graph database, while in other systems it costs a lot. We will further illustrate how this data model benefits our specific business questions in Section 2.

B.3 Node description:

The movies, actors (actresses), directors, and customers will be node labels. We only extract the properties that benefit our business questions from the raw datasets. Node *Customer* only has a property *id*. Node *Movie* has properties such as *id*, *name*, *genres*, and *ratings*. Node *Director* and *Actor* have *id* and *name* as their properties.

Table 1. Node description

Node label	Property	Type	Source	Description
Customer	id	string	MovieLens	Identify the customers Example: "c000001"
Movie	id	string	Multi	Identify the movie Example: "m172063"
	title	string array	Multi	Multiple movies can have the same name -> we use *id* as key Example: ["Carmencita"]
	year	integer	IMDb	Movie release year Example: 1894
	genres	string array	MovieLens	Movie genres: action, animation, comedy, crime, drama, experimental, fantasy, historical, horror, romance, scifi, thriller, western. A movie can be categorized into multiple genres. Example: ["action", "horror"]

	income	integer	IMDb	The revenue made by this movie (Unit: USD) Example: 110,450
	IMDb_rate	float	IMDb	IMDb ratings in the range of 0-10 Example: 5.7
	TMDb_rate	float	TMDb	TMDb ratings in the range of 0-100 Example: 52.0
Actor/Actress	id	string	IMDb	Identify the actor/actress Example: "nm00001"
	name	string	IMDb	Name of the actor/actress Multiple actors can have the same name -> we use *id* as key Example: "Fred Astaire"
Director	id	string	IMDb	Identify the director Example: "nm00001"
	name	string	IMDb	Name of the director Multiple directors can have the same name -> we use *id* as key Example: "Fred Astaire"

Note:

1. Since we have multiple datasets as our sources, we need to check consistency among them and we will illustrate how to import data specifically in Section 3.
2. We have 3 kinds of rating criterias from IMDb, TMDb and MovieLens. As we can only get average ratings for each movie from IMDb, TMDb, we use MovieLens ratings to form the *rate* relationship between customers and movies.

B.4 Link (Relationship) description:

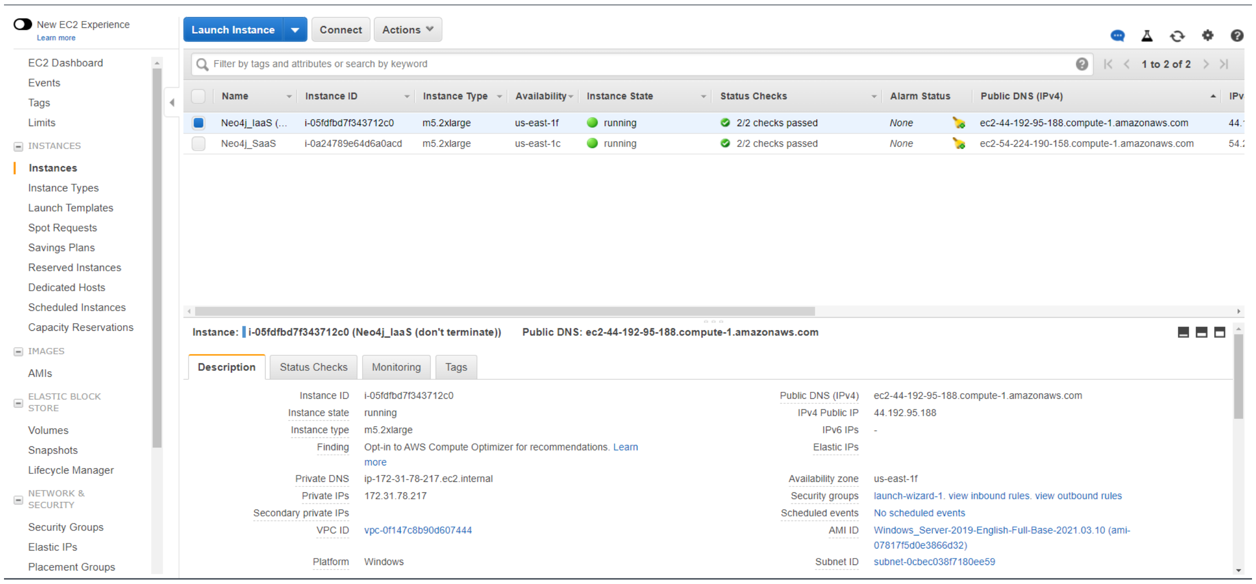
There will be three kinds of links. The links connecting customers and movies take the customers' rating score for the movie as the property. The links between movies and actors or directors show the crew relationship.

Table 2. Link description

Link	Property	Type	Source	Description
Rate (Customer ⟶ Movie)	ML_rate	float	MoieLens	Customer's rating for the movie represents a relationship between them. Example: 3.5
Acted in (Worker ⟶ Movie)	N/A	N/A	IMDb	The actor acts in the movie
Direct (Worker ⟶ Movie)	N/A	N/A	IMDb	The director directs the movie

Appendix C
Hardware and Software

We deploy the database on AWS as shown in the following figure.



Name	Instance ID	Instance Type	Availability	Instance State	Status Checks	Alarm Status	Public DNS (IPv4)	IPv
Neo4j_IaaS (...	i-05fdfbd7f343712c0	m5.2xlarge	us-east-1f	running	2/2 checks passed	None	ec2-44-192-95-188.compute-1.amazonaws.com	44.
Neo4j_SaaS	i-0a24789e64d6a0acd	m5.2xlarge	us-east-1c	running	2/2 checks passed	None	ec2-54-224-190-158.compute-1.amazonaws.com	54.

Instance: i-05fdfbd7f343712c0 (Neo4j_IaaS (don't terminate)) Public DNS: ec2-44-192-95-188.compute-1.amazonaws.com

Instance ID	i-05fdfbd7f343712c0	Public DNS (IPv4)	ec2-44-192-95-188.compute-1.amazonaws.com
Instance state	running	IPv4 Public IP	44.192.95.188
Instance type	m5.2xlarge	IPv6 IPs	-
Finding	Opt-in to AWS Compute Optimizer for recommendations. Learn more	Elastic IPs	
Private DNS	ip-172-31-78-217.ec2.internal	Availability zone	us-east-1f
Private IPs	172.31.78.217	Security groups	launch-wizard-1. view inbound rules. view outbound rules
Secondary private IPs		Scheduled events	No scheduled events
VPC ID	vpc-0f147c8b90d607444	AMI ID	Windows_Server-2019-English-Full-Base-2021.03.10 (ami-07817f5d0e3866d32)
Platform	Windows	Subnet ID	subnet-0cbec038f7180ee59

Figure C1. AWS EC2 machine management console

We choose the m5.2xlarge instance, whose configurations are shown in Table E1. Please check the Link for other instance configurations.

Table C1. Configuration of m5.2xlarge instance

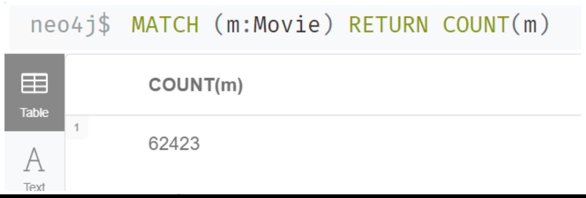
vCPU	Memory (GiB)	Network Bandwidth (Gbps)	EBS Bandwidth (Mbps)
8	32	Up to 10	Up to 4,750

Neo4j community server version: 4.1.0 (released on Jun 25, 2020). Please check the Link for more information.

Appendix D.
Screenshots of running the query code

Table 3. Questions about movie rating start with Index A. Questions about actor/director start with Index B. Questions about recommending the movie to customers start with Index C.

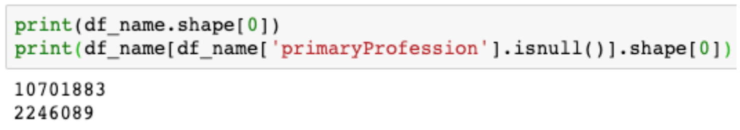
Easy	**A1. How many movies are there in the database?**



```
neo4j$ MATCH (m:Movie) RETURN COUNT(m)
```

	COUNT(m)
1	62423

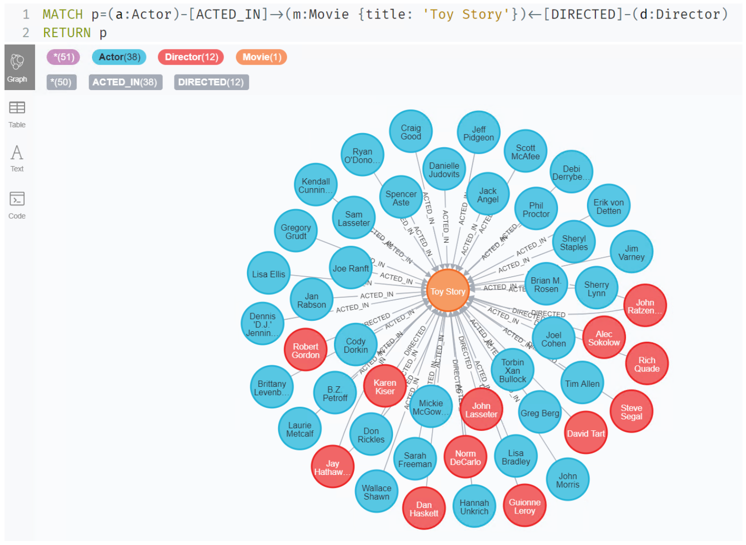
Easy	**A2. How many null values are there in the *primaryProfession* column in the cast name dataset?**



```
print(df_name.shape[0])
print(df_name[df_name['primaryProfession'].isnull()].shape[0])
```

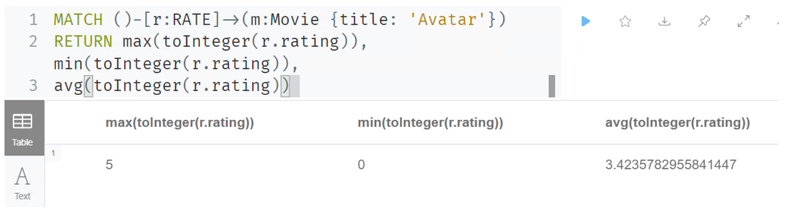
```
10701883
2246089
```

Easy	**A3: Who are the directors and casts known for the movie “Toy Story”?**



```
MATCH p=(a:Actor)-[ACTED_IN]→(m:Movie {title: 'Toy Story'})←[DIRECTED]-(d:Director)
RETURN p
```

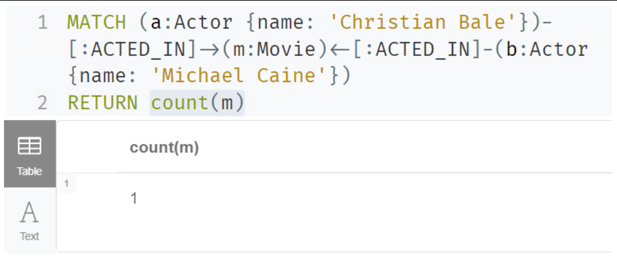
Easy	**A4. What is the max/min/avg of the MovieLens ratings for “Avatar”?**



```
MATCH ()-[r:RATE]→(m:Movie {title: 'Avatar'})
RETURN max(toInteger(r.rating)),
min(toInteger(r.rating)),
avg(toInteger(r.rating))
```

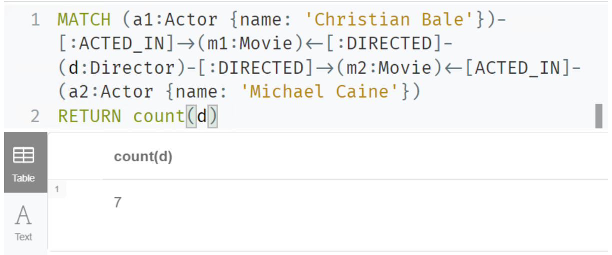
	max(toInteger(r.rating))	min(toInteger(r.rating))	avg(toInteger(r.rating))
1	5	0	3.4235782955841447

Moderate	**B1. How many movies Christian Bale and Michael Caine both acted in?**


```
MATCH (a:Actor {name: 'Christian Bale'})-
[:ACTED_IN]→(m:Movie)←[:ACTED_IN]-(b:Actor
{name: 'Michael Caine'})
RETURN count(m)
```

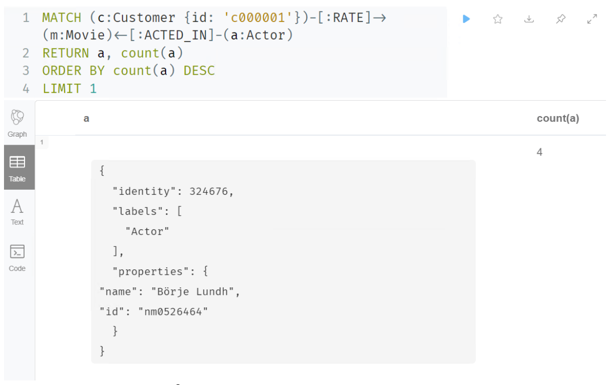
```
count(m)
1
```

Moderate	**B2. How many directors that both Christian Bale and Michael Caine worked with?**


```
MATCH (a1:Actor {name: 'Christian Bale'})-
[:ACTED_IN]→(m1:Movie)←[:DIRECTED]-
(d:Director)-[:DIRECTED]→(m2:Movie)←[ACTED_IN]-
(a2:Actor {name: 'Michael Caine'})
RETURN count(d)
```

```
count(d)
7
```

Moderate	**C1. Which actor/actresses does a specific customer probably like the most, who we ca recommend to this customer in the future, based on all the movies he/she rated?**

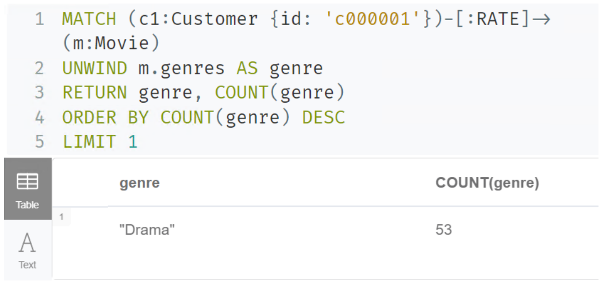

```
MATCH (c:Customer {id: 'c000001'})-[:RATE]→
(m:Movie)←[:ACTED_IN]-(a:Actor)
RETURN a, count(a)
ORDER BY count(a) DESC
LIMIT 1
```

```
a                                              count(a)
                                               4
{
  "identity": 324676,
  "labels": [
    "Actor"
  ],
  "properties": {
"name": "Börje Lundh",
"id": "nm0526464"
  }
}
```

Challenging	**C2. Which movie should be recommended to a specific customer?**

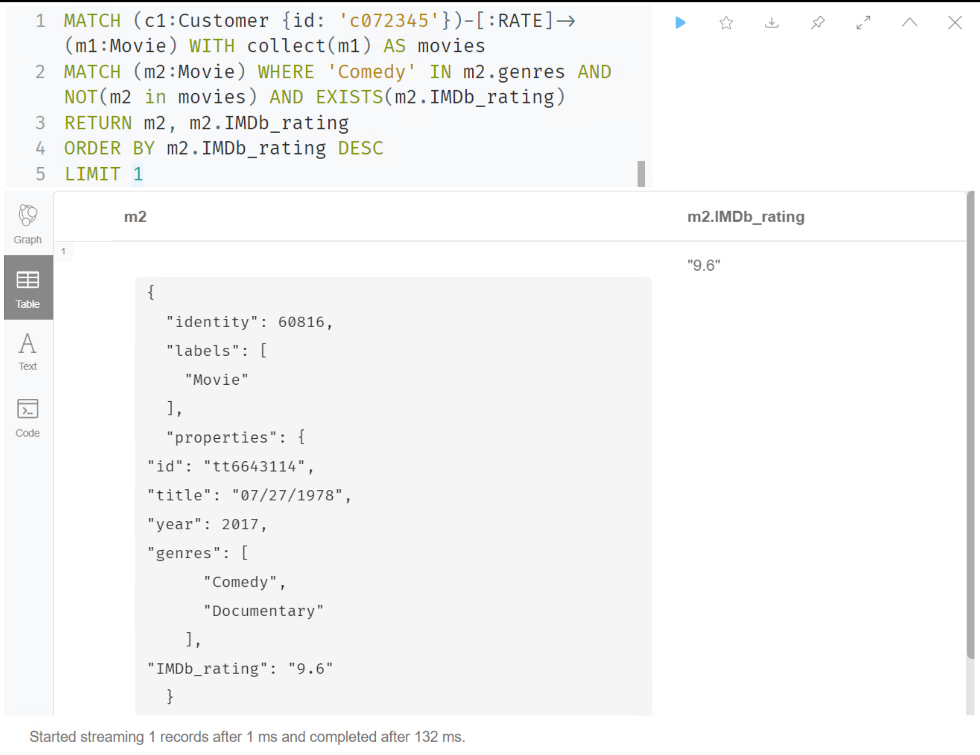
Method 1. Based on the movie genres

Step 1. Find the most frequent genre in the movie that the user has watched:


```
MATCH (c1:Customer {id: 'c000001'})-[:RATE]→
(m:Movie)
UNWIND m.genres AS genre
RETURN genre, COUNT(genre)
ORDER BY COUNT(genre) DESC
LIMIT 1
```

```
genre          COUNT(genre)
"Drama"        53
```

Step 2. Find the movie in this genre with the highest IMDb rating and this user hasn't watched:


```
MATCH (c1:Customer {id: 'c072345'})-[:RATE]→(m1:Movie) WITH collect(m1) AS movies
MATCH (m2:Movie) WHERE 'Comedy' IN m2.genres AND NOT(m2 in movies) AND EXISTS(m2.IMDb_rating)
RETURN m2, m2.IMDb_rating
ORDER BY m2.IMDb_rating DESC
LIMIT 1
```

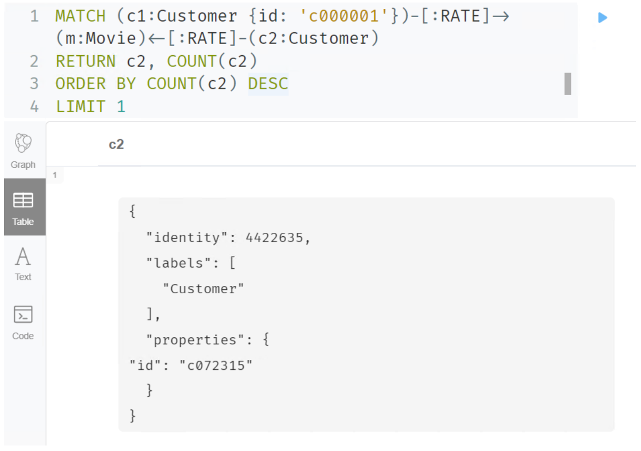
	m2	m2.IMDb_rating

1	{
	"identity": 60816,
	"labels": [
	"Movie"
	],
	"properties": {
	"id": "tt6643114",
	"title": "07/27/1978",
	"year": 2017,
	"genres": [
	"Comedy",
	"Documentary"
	],
	"IMDb_rating": "9.6"
	}
		"9.6"

Started streaming 1 records after 1 ms and completed after 132 ms.

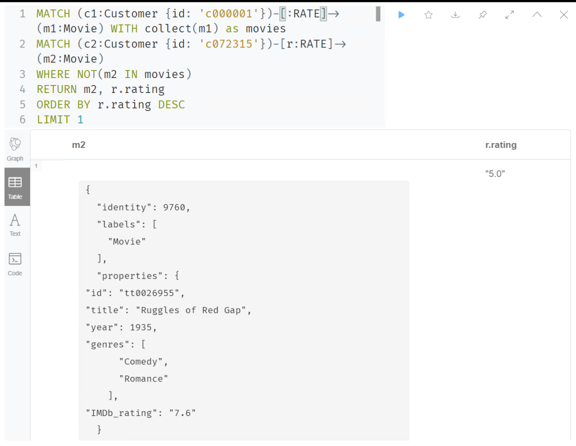
Method 2. Based on the similar user

Step 1. Based on the movie watching history to find the most similar person:


```
MATCH (c1:Customer {id: 'c000001'})-[:RATE]→(m:Movie)←[:RATE]-(c2:Customer)
RETURN c2, COUNT(c2)
ORDER BY COUNT(c2) DESC
LIMIT 1
```

```
c2
1
{
  "identity": 4422635,
  "labels": [
    "Customer"
  ],
  "properties": {
"id": "c072315"
  }
}
```

Step 2. Recommend the movie with the highest rating from that similar person:

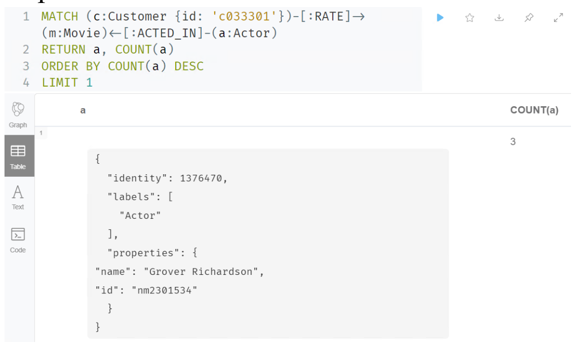


```
MATCH (c1:Customer {id: 'c000001'})-[:RATE]->
(m1:Movie) WITH collect(m1) as movies
MATCH (c2:Customer {id: 'c072315'})-[r:RATE]->
(m2:Movie)
WHERE NOT(m2 IN movies)
RETURN m2, r.rating
ORDER BY r.rating DESC
LIMIT 1
```

```
m2                                              r.rating
                                                "5.0"
  {
    "identity": 9760,
    "labels": [
      "Movie"
    ],
    "properties": {
  "id": "tt0026955",
  "title": "Ruggles of Red Gap",
  "year": 1935,
  "genres": [
        "Comedy",
        "Romance"
      ],
  "IMDb_rating": "7.6"
    }
```

Method 3. Based on the favorite actor / actress

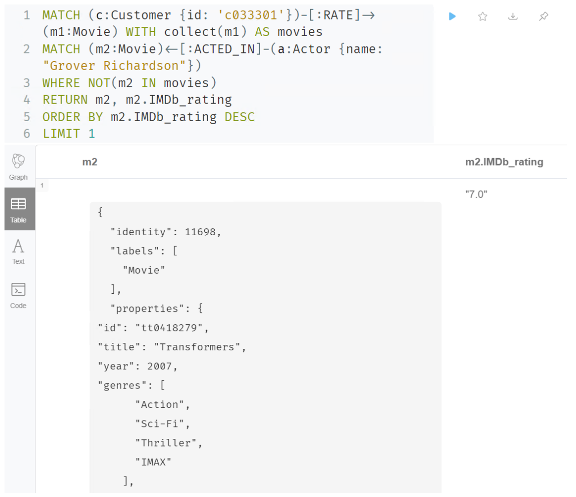
Step1. Find user's favorite actor:



```
MATCH (c:Customer {id: 'c033301'})-[:RATE]->
(m:Movie)<-[:ACTED_IN]-(a:Actor)
RETURN a, COUNT(a)
ORDER BY COUNT(a) DESC
LIMIT 1
```

```
a                                               COUNT(a)
                                                3
  {
    "identity": 1376470,
    "labels": [
      "Actor"
    ],
    "properties": {
  "name": "Grover Richardson",
  "id": "nm2301534"
    }
  }
```

Step2. Find the highest-rated IMDb movie that actor has played:



```
MATCH (c:Customer {id: 'c033301'})-[:RATE]->
(m1:Movie) WITH collect(m1) AS movies
MATCH (m2:Movie)<-[:ACTED_IN]-(a:Actor {name:
"Grover Richardson"})
WHERE NOT(m2 IN movies)
RETURN m2, m2.IMDb_rating
ORDER BY m2.IMDb_rating DESC
LIMIT 1
```

```
m2                                              m2.IMDb_rating
                                                "7.0"
  {
    "identity": 11698,
    "labels": [
      "Movie"
    ],
    "properties": {
  "id": "tt0418279",
  "title": "Transformers",
  "year": 2007,
  "genres": [
        "Action",
        "Sci-Fi",
        "Thriller",
        "IMAX"
      ],
```