# One Page Summary

# Topic: Movie Recommendation System Using Graph Database

Problem Statement:  The goal is to develop the model which will allow us to find movie recommendation based on user’s previous experience. I’ll use neo4J graph database as a tool. I will use 2 datasets: MovieLens, which contains ratings and tag applications applied toward movies by users, and TMDB 5000 Movie Dataset, which contains, among other, credits data for movies.

Dataset Description:

1. MovieLens (Small) contains 100,000 ratings and 3,600 tag applications applied to 9,000 movies by 600 users. Last updated 9/2018. Size: 1 MB.

<https://grouplens.org/datasets/movielens/>

1. TMDB 5000 Movie Dataset contains 2 csv files: one with detailed information about the movie (budget, genres, original language and so forth), the second one – contains movies credits – actors, directors, producers. Only csv file with credentials will be used. Size of the compressed tmdb\_5000\_credits.csv file: 7.64 MB.

<https://www.kaggle.com/tmdb/tmdb-movie-metadata>

Overview of Technology: neo4j is a graph-based database; Cypher is declarative graph query language; Python (via Jupiter notebook) was used only for preparing data.

## Overview of Steps:

1. Defined problem statement
2. Install and configure the environment
3. Find suitable dataset and obtain data
4. Preprocess data
5. Load data to a graph database
6. Find and evaluate multiple recommendation schemas.

## Hardware: PC with Windows 10 Home (64 bit) running on AMD FX-8320E Eight-Core Processor 3.5 GHz and equipment with 8.00 Gb RAM. No CUDA-supported GPU.

Software: neo4j, Python, Anaconda, Jupiter Notebook.

Lessons Learned: a model for movie recommendation system using previous user experience can be successfully and easily created using I used neo4j graph database and declarative graph query language Cypher. Neo4j fits perfectly for this task.

Pros: With graph database we have fast access to both data (user, movie, genre) and relationships between them, which allow us to process queries very fast, enabling using the model for real-time recommendation engines. Another advantage of using a graph database for this model is that it’s easy to visualize and understand the connections and paths with lead us to recommendations.

Cons: It’s hard to evaluate the performance of proposed models without access to real users in real time. It would be nice to expand the model by adding more features and connections.

# Problem Statement:

The goal is to develop the model which will allow us to find movie recommendation based on user’s previous experience. I’ll use neo4J graph database as a tool. I will use 2 datasets: MovieLens, which contains ratings and tag applications applied toward movies by users, and TMDB 5000 Movie Dataset, which contains, among other, credits data for movies.

# Why this topic?

I was interested in using the graph as a representation of data (nodes) and the connection between entities (edges). despite that I’ve never user Cypher query language and graph databases itself, I felt like it would be a useful experience. Amount different implementation of graph databased, I felt curiosity toward recommendation engines, as it is something that surrounds us everywhere in a modern digital word. It is a very useful technology for both providers (stores, online marketplaces, the music of movie aggregators) and users because it will provide them with more relevant content. Movies seem to be a consistent topic, compared to, for example, products sold at Amazon, as there are millions of good sold, and only tens of thousands of well-documented movies.

# Technologies used:



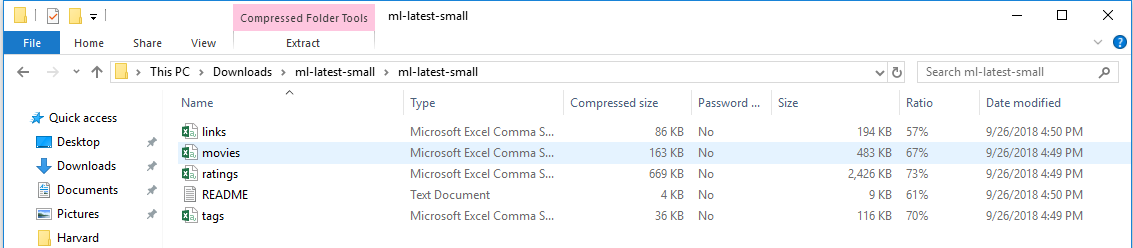


# Dataset Description:

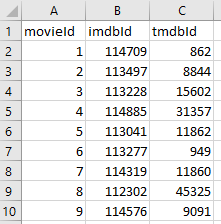
1. MovieLens (Small) dataset, according to its own description, describes 5-star rating and free-text tagging activity from [MovieLens]( <http://movielens.org> ), a movie recommendation service. It contains 100836 ratings and 3683 tag applications across 9742 movies. These data were created by 610 users between March 29, 1996 and September 24, 2018. This dataset was generated on September 26, 2018.

<https://grouplens.org/datasets/movielens/>

Here how does the downloaded zip file looks like:



File Links.csv contains 3 different ids of each movie: movieId – the one used in MovieLens dataset, imdbId – is corresponding to IMDB dataset and tmdbId – id corresponding to tmdb <https://www.themoviedb.org/> dataset, which we’ll use to get information about actors and directors of the movies. E.g., the movie Toy Story has the link <https://www.themoviedb.org/movie/862>.



File movies.csv has information about movie id, the title along with the year of release in parentheses, and genres, separated by “|” , which selected from the following list:

\* Action

\* Adventure

\* Animation

\* Children's

\* Comedy

\* Crime

\* Documentary

\* Drama

\* Fantasy

\* Film-Noir

\* Horror

\* Musical

\* Mystery

\* Romance

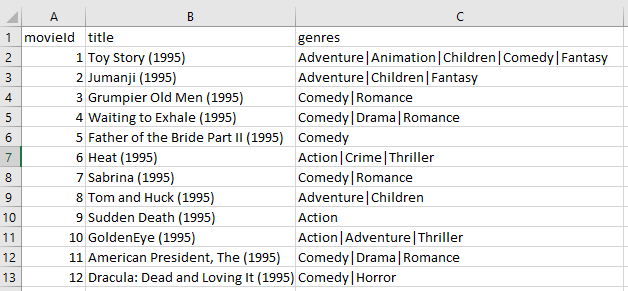
\* Sci-Fi

\* Thriller

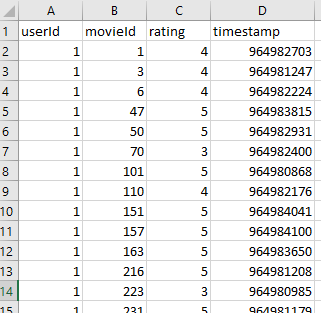
\* War

\* Western

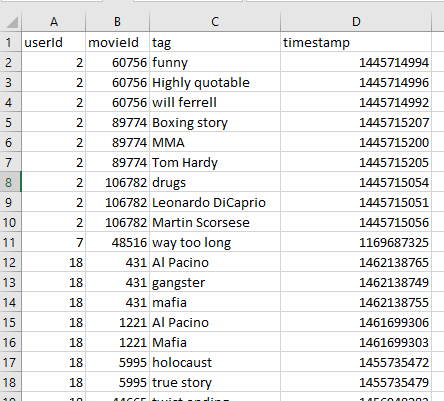
\* (no genres listed)



Each line of file ratings.csv contains rating made on a 5-star scale, with half-star increments (0.5 stars - 5.0 stars) of one movie by one user. The user is represented by id only. User ids have been anonymized.



Each row of file tags.csv represents one tag applied to one movie by one user. Tags are user-generated metadata about movies.



1. TMDB 5000 Movie Dataset. This dataset was generated from The Movie Database API. It contains 2 csv files: one with detailed information about the movie (budget, genres, original language and so forth), the second one – contains movies credits – actors, directors, producers. Only one of two csv files with credentials will be used.

Compressed size of tmdb\_5000\_credits.csv file: 7.64 MB; uncompressed: 39 MB.

<https://www.kaggle.com/tmdb/tmdb-movie-metadata>

Each row of this file contains movie\_id, title, cast and crew information.

We will use only actors name’s from cast column and directors from crew column.

East cell of “cast” column contains JSON formatted data with looks like follows:

[

{

"cast\_id": 4,

"character": "Captain Jack Sparrow",

"credit\_id": "52fe4232c3a36847f800b50d",

"gender": 2,

"id": 85,

"name": "Johnny Depp",

"order": 0

},

{

"cast\_id": 5,

"character": "Will Turner",

"credit\_id": "52fe4232c3a36847f800b511",

"gender": 2,

"id": 114,

"name": "Orlando Bloom",

"order": 1

},

…

Cells of “crew” column:

[

{

"credit\_id": "52fe4273c3a36847f801fa8d",

"department": "Writing",

"gender": 1,

"id": 10966,

"job": "Novel",

"name": "J.K. Rowling"

},

{

"credit\_id": "52fe4273c3a36847f801fa81",

"department": "Directing",

"gender": 2,

"id": 11343,

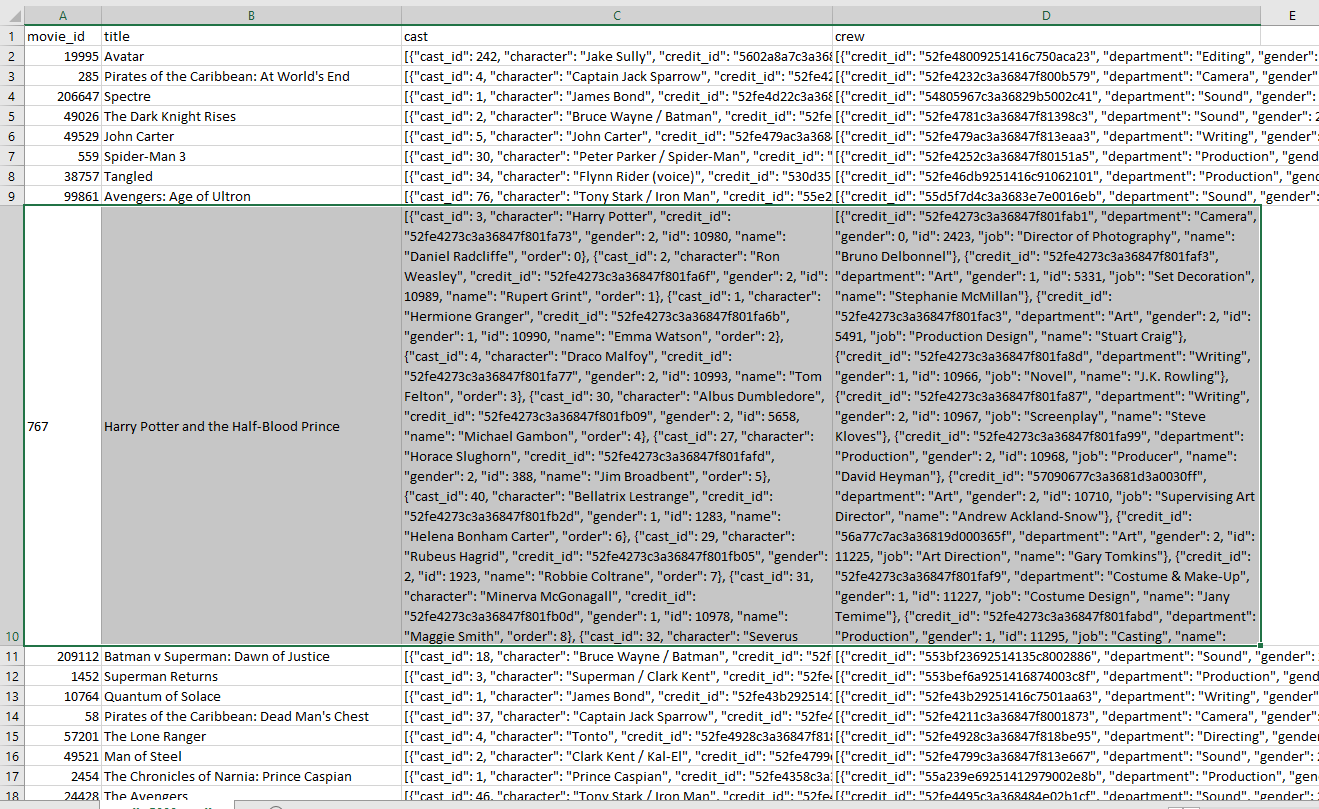
"job": "Director",

"name": "David Yates"

},

…

Content of entire file looks like follows:

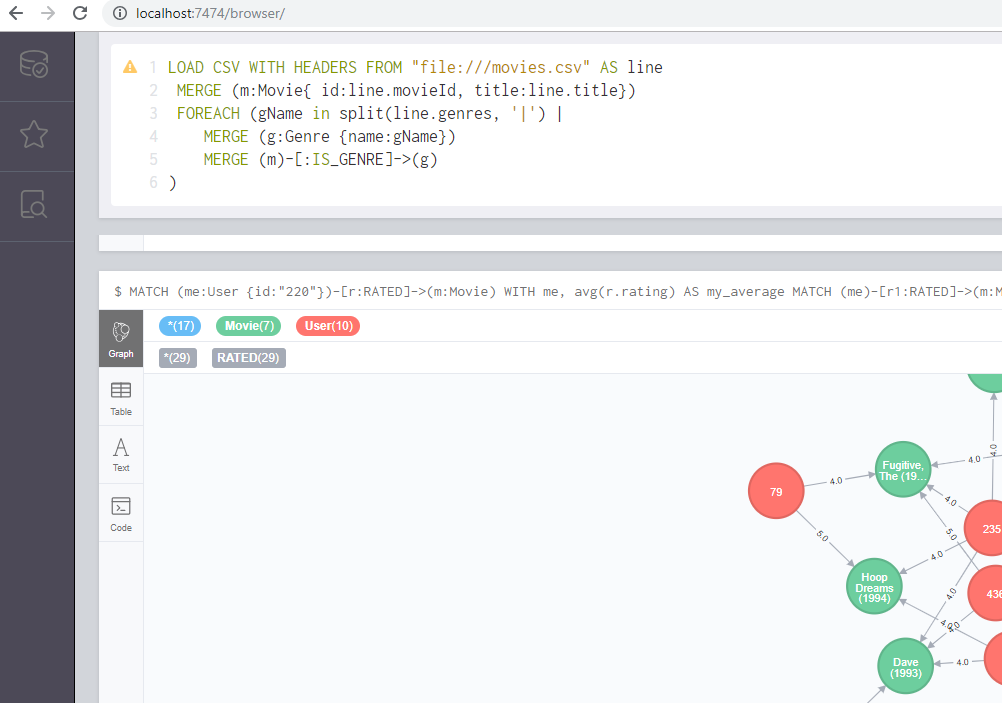


# Data reading and preprocessing:

All queries have been written using Cypher language.

To reproduce results, please go to <http://localhost:7474/browser/>

and simply copy queries from this report to the command line at the top of the page:



Data from MovieLens dataset can be easily downloaded to neo4j database using LOAD CSV function.

First, I placed csv files to %NEO4J\_HOME%/import folder.

Let’s download movie information by creating label Movie with properties id and title and label Genre with the single property title:

LOAD CSV WITH HEADERS FROM "[file:///movies.csv](file:///C:\movies.csv)" AS line

MERGE (m:Movie{ id:line.movieId, title:line.title})

FOREACH (gName in split(line.genres, '|') |

MERGE (g:Genre {name:gName})

MERGE (m)-[:IS\_GENRE]->(g)

)

Added 9762 labels, created 9762 nodes, set 19504 properties, created 22084 relationships, completed after 38484 ms.

By downloading data from ratings.csv we will create label User with only property id (because data about users are anonymized) and connection RATED with property rating: (User)-[:RATED { rating:}]->(Movie).

LOAD CSV WITH HEADERS FROM "[file:///ratings.csv](file:///C:\ratings.csv)" AS line

MATCH (m:Movie {id:line.movieId})

MERGE (u:User {id:line.userId})

MERGE (u)-[:RATED { rating: toFloat(line.rating)}]->(m);

Added 610 labels, created 610 nodes, set 101446 properties, created 100836 relationships, completed after 695343 ms.

Tags:

LOAD CSV WITH HEADERS FROM "[file:///tags.csv](file:///C:\tags.csv)" AS line

MATCH (m:Movie {id:line.movieId})

MATCH (u:User {id:line.userId})

CREATE (u)-[:TAGGED { tag: line.tag}]->(m);

Set 3683 properties, created 3683 relationships, completed after 21062 ms.

As described above, file Links.csv contains 3 different ids of each movie: movieId – the one used in MovieLens dataset, imdbId – is corresponding to IMDB dataset and tmdbId – id corresponding to tmdb <https://www.themoviedb.org/> dataset, which we’ll use to get information about actors and directors of the movies. Let’s add to each movie new property – tmdbId:

LOAD CSV WITH HEADERS FROM "[file:///links.csv](file:///C:\links.csv)" AS line

MATCH (m:Movie {id:line.movieId})

SET m.tmdbId=line.tmdbId;

Set 9734 properties, completed after 63423 ms.

Now let’s proceed with information about actors and directors. As content of tmdb\_5000\_credits.csv is not that easy to download to neo4j (csv with JSON format for some columns content) and, taking into the consideration that we don’t need all the information from this file (we will not use, for example, information about Director of Photography of Casting Director to make a recommendation, with all the respect to them), let’s create simple Python application with will read all the info from tmdb\_5000\_credits.csv file, filter it, and create csv file with easy to read for neo4j data.

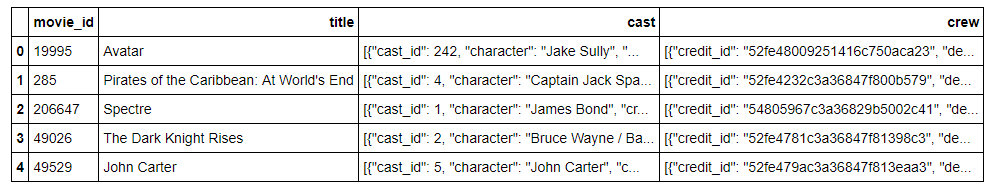
First, let’s read data from csv file to Pandas dataframe:

**import** **pandas** **as** **pd**

**import** **json**

data = pd.read\_csv("C:**\\**Users**\\**Aleks**\\**Desktop**\\**BD final**\\**tmdb\_5000\_credits.csv")

data.head()



Let’s examine “cast” column. East cell contains JSON formatted data with looks like follows:

[

{

"cast\_id": 4,

"character": "Captain Jack Sparrow",

"credit\_id": "52fe4232c3a36847f800b50d",

"gender": 2,

"id": 85,

"name": "Johnny Depp",

"order": 0

},

{

"cast\_id": 5,

"character": "Will Turner",

"credit\_id": "52fe4232c3a36847f800b511",

"gender": 2,

"id": 114,

"name": "Orlando Bloom",

"order": 1

},

{

"cast\_id": 6,

"character": "Elizabeth Swann",

"credit\_id": "52fe4232c3a36847f800b515",

"gender": 1,

"id": 116,

"name": "Keira Knightley",

"order": 2

},

…

We need only actor’s name and role. Let’s read the data we need to new dataframe:

castDf = pd.DataFrame({'movieId':[], 'person\_name':[], 'role':[]})

**for** index, row **in** data.iterrows():

movieId = row['movie\_id']

c = pd.DataFrame.from\_dict(json.loads(row['cast']))

**for** index, row **in** c.iterrows():

castDf.loc[len(castDf)] = [str(movieId), row['name'], row['character']]

castDf.count()

movieId 106257

person\_name 106257

role 106257

dtype: int64

We don’t need that much of actors. Most of them probably plays once, in role like Waitress. Let’s remove those who played less than 5 times, as they will unlike be helpful in movie recommendations:

castDf['count'] = castDf.groupby('person\_name')['person\_name'].transform(pd.Series.value\_counts)

castDf = castDf[castDf['count']>5]

castDf.drop('count', axis=1, inplace=**True**)

castDf.count()

movieId 33470

person\_name 33470

role 33470

dtype: int64

So we’ll proceed with 33 thousand actors instead of 106 thousand.

Let’s examine crew column:

[

{

"credit\_id": "52fe4273c3a36847f801fab1",

"department": "Camera",

"gender": 0,

"id": 2423,

"job": "Director of Photography",

"name": "Bruno Delbonnel"

},

{

"credit\_id": "52fe4273c3a36847f801fa8d",

"department": "Writing",

"gender": 1,

"id": 10966,

"job": "Novel",

"name": "J.K. Rowling"

},

{

"credit\_id": "52fe4273c3a36847f801fa81",

"department": "Directing",

"gender": 2,

"id": 11343,

"job": "Director",

"name": "David Yates"

},

…

I’ll use only information about directors:

directorDf = pd.DataFrame({'movieId':[], 'person\_name':[]})

**for** index, row **in** data.iterrows():

movieId = row['movie\_id']

crew = pd.DataFrame.from\_dict(json.loads(row['crew']))

**if** (**not** (crew.empty)):

nameList = crew[crew['job']=='Director']['name'].values

**if** (len(nameList)>0):

directorDf.loc[len(directorDf)] = [str(movieId), nameList[0]]

directorDf.count()

movieId 4773

person\_name 4773

dtype: int64

Based on same logic as with actors, let’s discard those who directed less than 3 movies, as it wouldn’t be much helpful for recommendations:

directorDf['count'] = directorDf.groupby('person\_name')['person\_name'].transform(pd.Series.value\_counts)

directorDf = directorDf[directorDf['count']>3]

directorDf.drop('count', axis=1, inplace=**True**)

directorDf.count()

movieId 2058

person\_name 2058

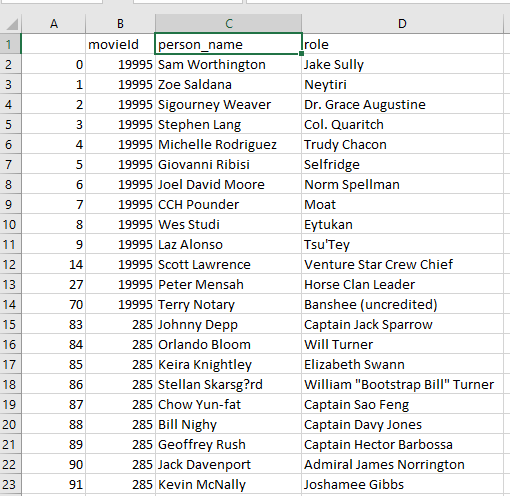
dtype: int64

Now let’s write obtained dataframes to csv file:

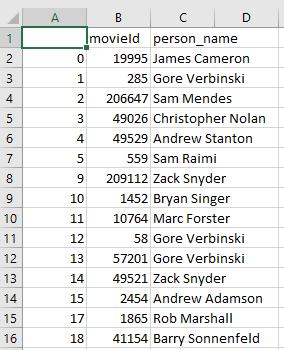
castDf.to\_csv("C:**\\**Users**\\**Aleks**\\**Desktop**\\**BD final**\\**roles.csv")

directorDf.to\_csv("C:**\\**Users**\\**Aleks**\\**Desktop**\\**BD final**\\**directors.csv")

The content of file roles.csv:



And directors.csv:



Now we can go back to neo4j and read data about directors and actors:

LOAD CSV WITH HEADERS FROM "[file:///directors.csv](file:///C:\directors.csv)" AS line

MATCH (m:Movie{ tmdbId:line.movieId})

MERGE (p:Person{name:line.person\_name})

MERGE (p)-[:DIRECTED]->(m);

Added 339 labels, created 339 nodes, set 339 properties, created 1889 relationships, completed after 13784 ms.

LOAD CSV WITH HEADERS FROM "[file:///roles.csv](file:///C:\roles.csv)" AS line

MATCH (m:Movie{ tmdbId:line.movieId})

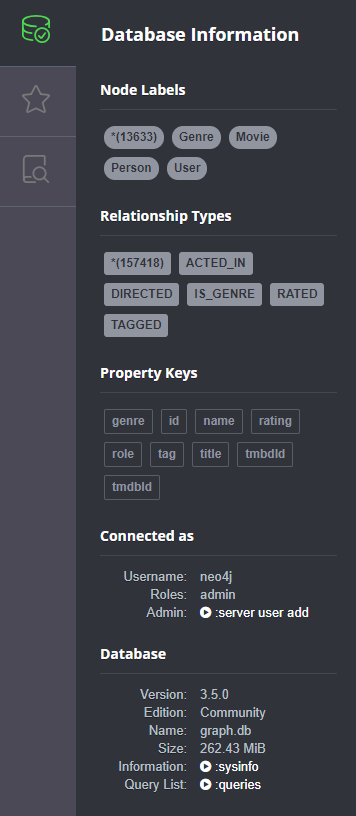
MERGE (p:Person{name:line.person\_name})

CREATE (p)-[r:ACTED\_IN] ->(m)

SET r.role= line.role;

Added 2922 labels, created 2922 nodes, set 31782 properties, created 28926 relationships, completed after 279049 ms.

Now, when all data have been read, let’s review general information about the obtained database:



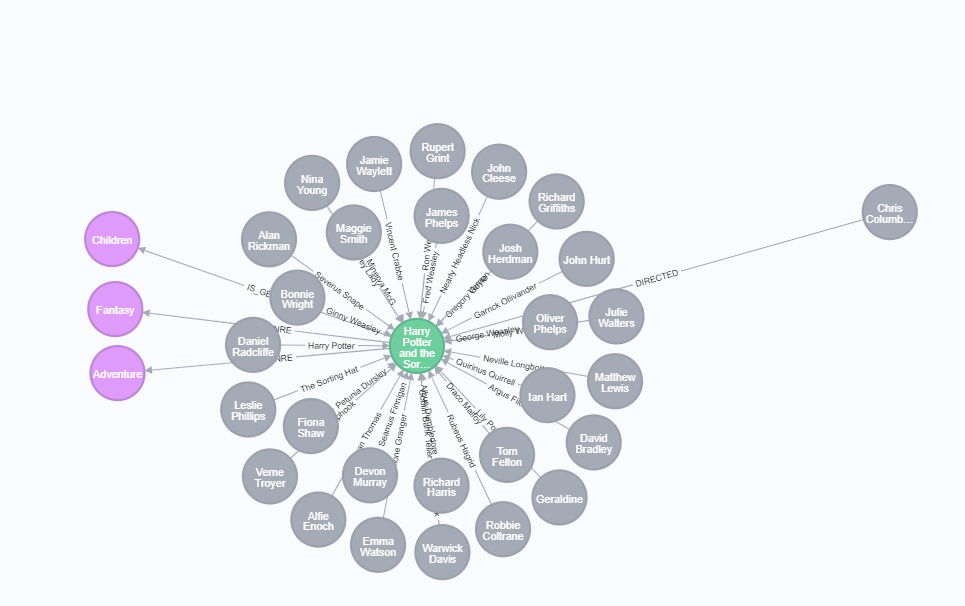
# Building recommendations

Let’s examine how our data looks like.

All genres, actors and director of a movie:

MATCH (m:Movie {title: "Harry Potter and the Sorcerer's Stone (a.k.a. Harry Potter and the Philosopher's Stone) (2001)"})-[:ACTED\_IN|:IS\_GENRE|:DIRECTED]-(p)

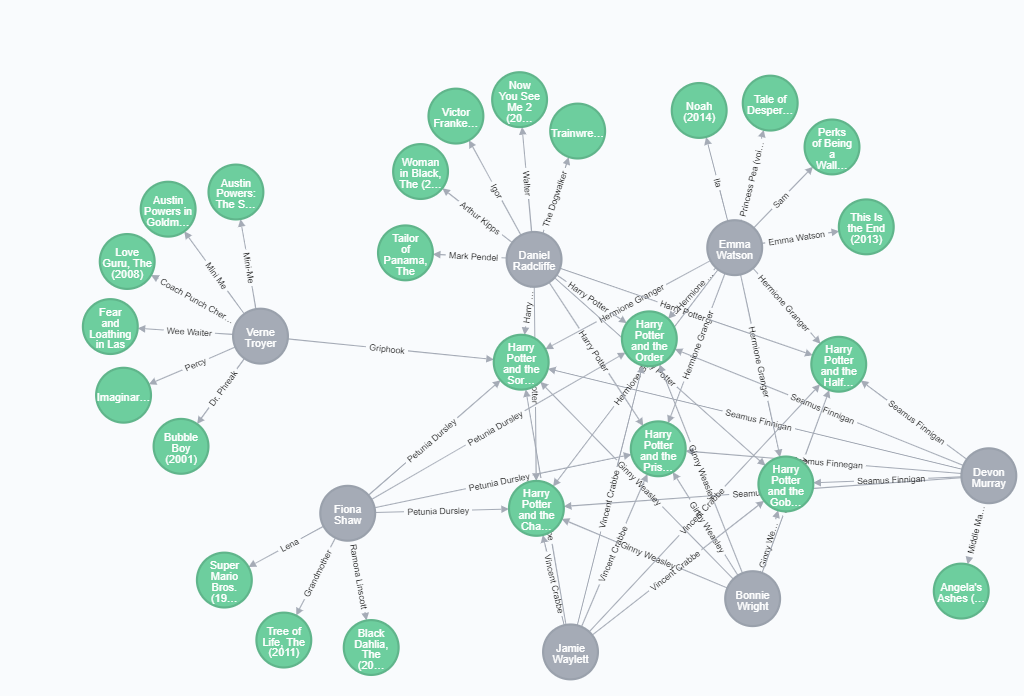
RETURN m, p



Movies with shared actors or directors (connected thought 2nd-degree connection):

MATCH q=(m:Movie {title: "Harry Potter and the Sorcerer's Stone (a.k.a. Harry Potter and the Philosopher's Stone) (2001)"})-[:ACTED\_IN |:DIRECTED\*..2]-(p)

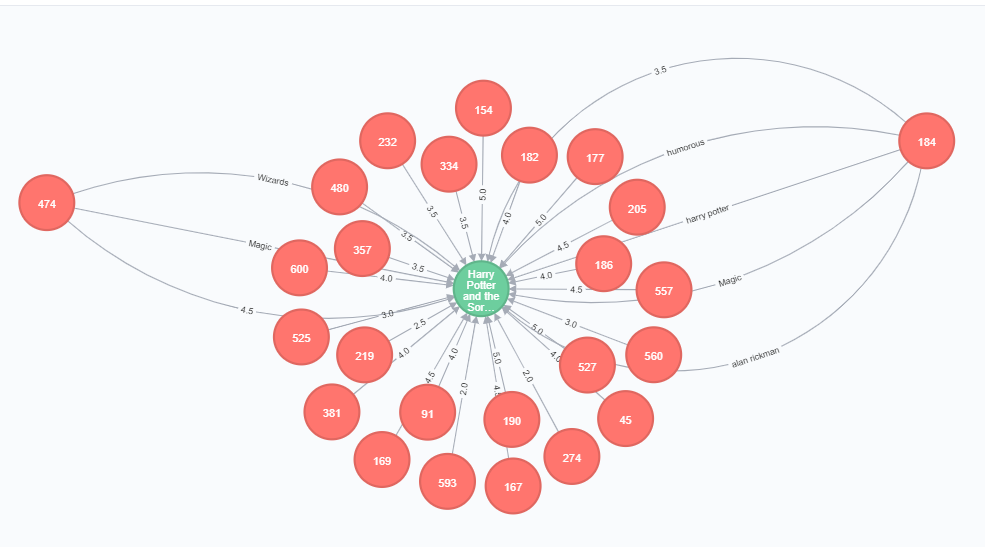
RETURN q LIMIT 50



Users who rated or tagged this movie:

MATCH (m:Movie {title: "Harry Potter and the Sorcerer's Stone (a.k.a. Harry Potter and the Philosopher's Stone) (2001)"})-[:RATED|:TAGGED]-(u)

RETURN m, u LIMIT 25



Now, when we are familiar with data, let’s build find some recommendation, starting with the simpliest one and gradually increasing the complexity of our queries.

The approach when we are taking in consideration only what other users liked is called **Collaborative Filtering.**

Let’s find movies targeted user likes, then find users who also liked that movies, and recommend movies that other users liked but which our user haven’t seen (rated), sorted by the number of “paths” that led to a particular recommendation.

MATCH (me:User{id:'220'})-[r1:RATED]->(m:Movie)<-[r2:RATED]-(other:User)-[r3:RATED]->(m2:Movie)

WHERE r1.rating > 3 AND r2.rating > 3 AND r3.rating > 3 AND NOT (me)-[:RATED]->(m2)

RETURN distinct m2 AS recommended\_movie, count(\*) AS score

ORDER BY score DESC

LIMIT 15

╒══════════════════════════════════════════════════════════════════════╤═══════╕

│"recommended\_movie" │"score"│

╞══════════════════════════════════════════════════════════════════════╪═══════╡

│{"title":"Silence of the Lambs, The (1991)","tmdbId":"274","id":"593"}│7203 │

├──────────────────────────────────────────────────────────────────────┼───────┤

│{"title":"Lord of the Rings: The Fellowship of the Ring, The (2001)","│6563 │

│tmdbId":"120","id":"4993"} │ │

├──────────────────────────────────────────────────────────────────────┼───────┤

│{"title":"American Beauty (1999)","tmdbId":"14","id":"2858"} │6227 │

├──────────────────────────────────────────────────────────────────────┼───────┤

│{"title":"Braveheart (1995)","tmdbId":"197","id":"110"} │5894 │

├──────────────────────────────────────────────────────────────────────┼───────┤

│{"title":"Gladiator (2000)","tmdbId":"98","id":"3578"} │5777 │

├──────────────────────────────────────────────────────────────────────┼───────┤

│{"title":"Schindler's List (1993)","tmdbId":"424","id":"527"} │5663 │

├──────────────────────────────────────────────────────────────────────┼───────┤

│{"title":"Monty Python and the Holy Grail (1975)","tmdbId":"762","id":│5377 │

│"1136"} │ │

├──────────────────────────────────────────────────────────────────────┼───────┤

│{"title":"Ocean's Eleven (2001)","tmdbId":"161","id":"4963"} │5011 │

├──────────────────────────────────────────────────────────────────────┼───────┤

│{"title":"Alien (1979)","tmdbId":"348","id":"1214"} │4951 │

├──────────────────────────────────────────────────────────────────────┼───────┤

As every tends to give more higher or lower ratings in general, let’s filter by average rating of particular user, rather than just constant “3”:

MATCH (me:User{id:'220'})-[r:RATED]-(m)

WITH me, avg(r.rating) AS average

MATCH (me)-[r1:RATED]->(m:Movie)<-[r2:RATED]-(other:User)-[r3:RATED]->(m2:Movie)

WHERE r1.rating > average AND r2.rating > average AND r3.rating > average AND NOT (me)-[:RATED]->(m2)

RETURN distinct m2 AS recommended\_movie, count(\*) AS score

ORDER BY score DESC

LIMIT 15

╒══════════════════════════════════════════════════════════════════════╤═══════╕

│"recommended\_movie" │"score"│

╞══════════════════════════════════════════════════════════════════════╪═══════╡

│{"title":"Silence of the Lambs, The (1991)","tmdbId":"274","id":"593"}│5322 │

├──────────────────────────────────────────────────────────────────────┼───────┤

│{"title":"Lord of the Rings: The Fellowship of the Ring, The (2001)","│4276 │

│tmdbId":"120","id":"4993"} │ │

├──────────────────────────────────────────────────────────────────────┼───────┤

│{"title":"American Beauty (1999)","tmdbId":"14","id":"2858"} │4129 │

├──────────────────────────────────────────────────────────────────────┼───────┤

│{"title":"Schindler's List (1993)","tmdbId":"424","id":"527"} │4086 │

├──────────────────────────────────────────────────────────────────────┼───────┤

│{"title":"Braveheart (1995)","tmdbId":"197","id":"110"} │3982 │

├──────────────────────────────────────────────────────────────────────┼───────┤

│{"title":"Gladiator (2000)","tmdbId":"98","id":"3578"} │3537 │

├──────────────────────────────────────────────────────────────────────┼───────┤

│{"title":"Alien (1979)","tmdbId":"348","id":"1214"} │3502 │

├──────────────────────────────────────────────────────────────────────┼───────┤

│{"title":"Monty Python and the Holy Grail (1975)","tmdbId":"762","id":│3408 │

│"1136"} │ │

├──────────────────────────────────────────────────────────────────────┼───────┤

│{"title":"Godfather: Part II, The (1974)","tmdbId":"240","id":"1221"} │3330 │

├──────────────────────────────────────────────────────────────────────┼───────┤

Here is visualization of some connections in previous query:

MATCH (me:User{id:'220'})-[r:RATED]-(m)

WITH me, avg(r.rating) AS average

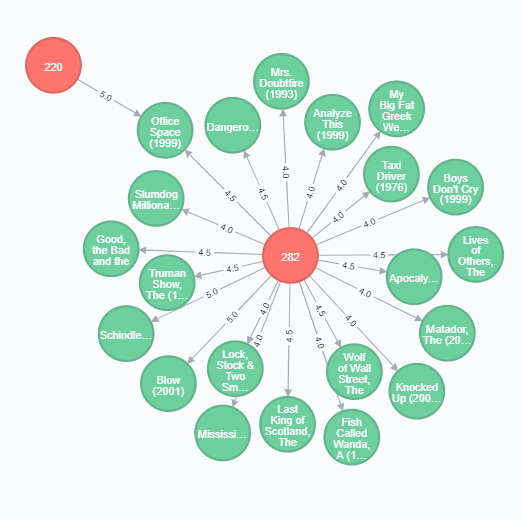
MATCH (me)-[r1:RATED]->(m:Movie)<-[r2:RATED]-(other:User)-[r3:RATED]->(m2:Movie)

WHERE r1.rating > average AND r2.rating > average AND r3.rating > average AND NOT (me)-[:RATED]->(m2)

RETURN distinct m2, other, me, m AS recommended\_movie, count(m2) AS score

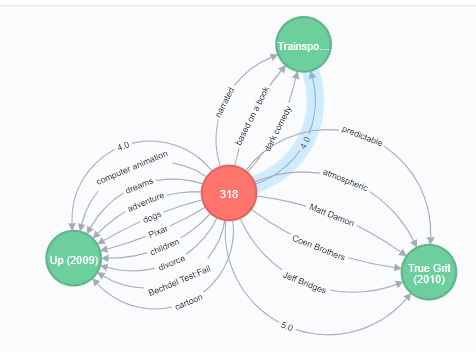
ORDER BY score DESC

LIMIT 20



Let’s use tags: find tags our user gave o describe movies he likes, and find other movies with same tags (not taking into consideration whether other users, who describe other movies liked that movies or not).

Here are the movies and tags of movies of our user’s liking:



MATCH (me:User{id:'318'})-[r:RATED]-(m)

WITH me, avg(r.rating) AS average

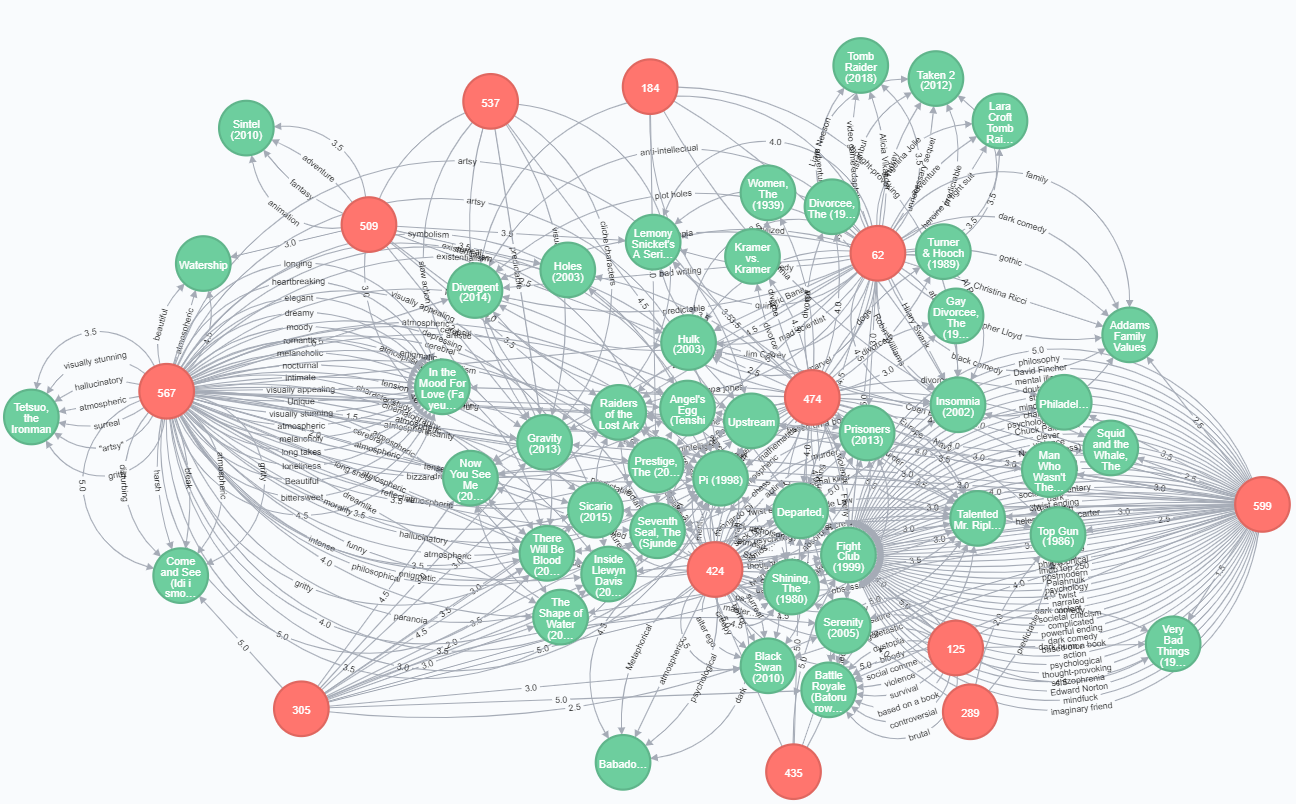
MATCH (me)-[t1:TAGGED]->(m:Movie)-[r:RATED]-(me)

MATCH (other:User)-[t2:TAGGED]->(m1:Movie)

WHERE r.rating > average AND t1.tag=t2.tag AND NOT (me)-[:TAGGED]->(m1) AND NOT (me)-[:RATED]->(m1)

RETURN m1, other





Every movie in this subgraph contains a tag our user liked.

Now let’s use collaborative approach together with information about the content of the movie (we have actors, directors, and genre).

First, let’s found actors on movies which our user liked sorted by the number of time particular actor appears in such movies:

MATCH (me:User{id:'318'})-[r:RATED]-(m:Movie)

WITH me, avg(r.rating) AS average

MATCH (me)-[r:RATED]->(m:Movie)-[:ACTED\_IN]-(p:Person)

WHERE r.rating > average

RETURN p as actor, COUNT(\*) AS score

ORDER BY score DESC LIMIT 10

╒═════════════════════════════════╤═══════╕

│"actor" │"score"│

╞═════════════════════════════════╪═══════╡

│{"name":"Johnny Depp"} │10 │

├─────────────────────────────────┼───────┤

│{"name":"Matt Damon"} │10 │

├─────────────────────────────────┼───────┤

│{"name":"George Clooney"} │9 │

├─────────────────────────────────┼───────┤

│{"name":"Bill Hader"} │8 │

├─────────────────────────────────┼───────┤

│{"name":"Brad Pitt"} │8 │

├─────────────────────────────────┼───────┤

│{"name":"Steve Buscemi"} │8 │

├─────────────────────────────────┼───────┤

│{"name":"John C. Reilly"} │7 │

├─────────────────────────────────┼───────┤

│{"name":"Philip Seymour Hoffman"}│7 │

├─────────────────────────────────┼───────┤

│{"name":"Sean Penn"} │7 │

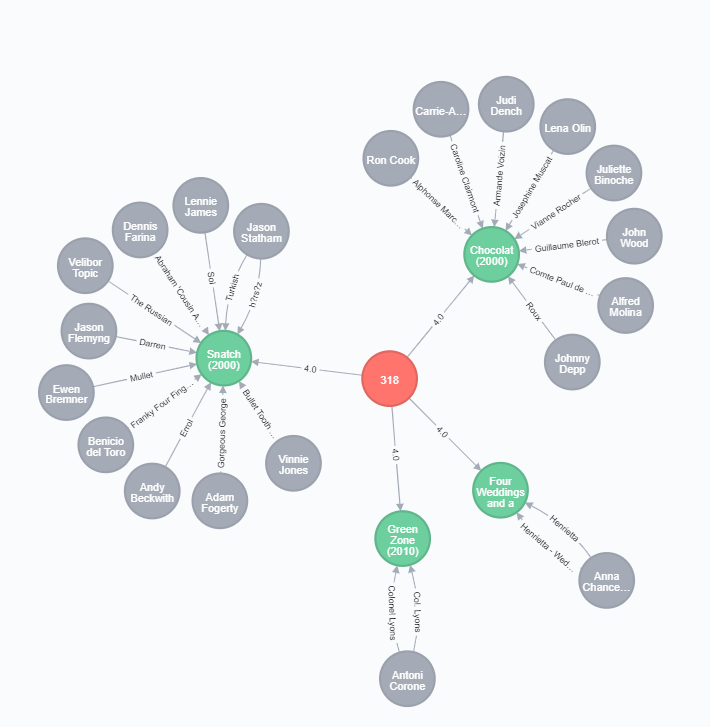
├─────────────────────────────────┼───────┤

│{"name":"Josh Brolin"} │6 │

└─────────────────────────────────┴───────┘

Apparently, our user #318 likes Johnny Depp.

Here is illustration:



Same with directors:

MATCH (me:User{id:'318'})-[r:RATED]-(m:Movie)

WITH me, avg(r.rating) AS average

MATCH (me)-[r:RATED]->(m:Movie)-[:DIRECTED]-(p:Person)

WHERE r.rating > average

RETURN p as director, COUNT(\*) AS score

ORDER BY score DESC LIMIT 10

╒════════════════════════════╤═══════╕

│"director" │"score"│

╞════════════════════════════╪═══════╡

│{"name":"Joel Coen"} │7 │

├────────────────────────────┼───────┤

│{"name":"Christopher Nolan"}│5 │

├────────────────────────────┼───────┤

│{"name":"Steven Spielberg"} │5 │

├────────────────────────────┼───────┤

│{"name":"Quentin Tarantino"}│4 │

├────────────────────────────┼───────┤

│{"name":"Kevin Smith"} │4 │

├────────────────────────────┼───────┤

│{"name":"Guy Ritchie"} │3 │

├────────────────────────────┼───────┤

│{"name":"Larry Charles"} │3 │

├────────────────────────────┼───────┤

│{"name":"Spike Lee"} │3 │

├────────────────────────────┼───────┤

│{"name":"Spike Jonze"} │3 │

├────────────────────────────┼───────┤

│{"name":"Jason Reitman"} │3 │

└────────────────────────────┴───────┘

And genres:

MATCH (me:User{id:'318'})-[r:RATED]-(m:Movie)

WITH me, avg(r.rating) AS average

MATCH (me)-[r:RATED]->(m:Movie)-[:IS\_GENRE]-(p:Genre)

WHERE r.rating > average

RETURN p.title as genre, COUNT(\*) AS score

ORDER BY score DESC LIMIT 10

╒═════════════╤═══════╕

│"genre" │"score"│

╞═════════════╪═══════╡

│"Drama" │232 │

├─────────────┼───────┤

│"Comedy" │150 │

├─────────────┼───────┤

│"Thriller" │77 │

├─────────────┼───────┤

│"Action" │73 │

├─────────────┼───────┤

│"Crime" │72 │

├─────────────┼───────┤

│"Adventure" │66 │

├─────────────┼───────┤

│"Documentary"│61 │

├─────────────┼───────┤

│"Romance" │49 │

├─────────────┼───────┤

│"Sci-Fi" │49 │

├─────────────┼───────┤

│"Animation" │47 │

└─────────────┴───────┘

Now let’s use combined information about favorite actors, directors and genres to provide user with weighted recommendation sorted by number of overlapping paths that lead to particular recommended movie:

MATCH (me:User{id:'318'})-[r:RATED]-(m:Movie)

WITH me, avg(r.rating) AS average

MATCH (me)-[r:RATED]->(m:Movie)

WHERE r.rating > average

MATCH (m)-[:IS\_GENRE]->(g:Genre)<-[:IS\_GENRE]-(rm:Movie)

WITH me, m, rm, COUNT(\*) AS gs

OPTIONAL MATCH (m)<-[:ACTED\_IN]-(a:Person)-[:ACTED\_IN]->(rm)

WITH me, m, rm, gs, COUNT(a) AS as

OPTIONAL MATCH (m)<-[:DIRECTED]-(d:Person)-[:DIRECTED]->(rm)

WITH me, m, rm, gs, as, COUNT(d) AS ds

MATCH (rm)

WHERE NOT (me)-[:RATED]->(rm)

RETURN rm.title AS recommendation,

gs as genre\_score, as as actor\_score, ds as director\_score,

(5\*gs)+(2\*as)+(5\*ds) AS weighed\_score

ORDER BY weighed\_score DESC LIMIT 10

5, 2, 5 are parameters we can adjust if we want to give more weight to either of categories.

╒══════════════════════════════════════════════╤═════════════╤═════════════╤════════════════╤═══════════════╕

│"recommendation" │"genre\_score"│"actor\_score"│"director\_score"│"weighed\_score"│

╞══════════════════════════════════════════════╪═════════════╪═════════════╪════════════════╪═══════════════╡

│"Toy Story 3 (2010)" │5 │11 │0 │47 │

├──────────────────────────────────────────────┼─────────────┼─────────────┼────────────────┼───────────────┤

│"The Hunger Games: Mockingjay - Part 2 (2015)"│2 │13 │1 │41 │

├──────────────────────────────────────────────┼─────────────┼─────────────┼────────────────┼───────────────┤

│"Kung Fu Panda 3 (2016)" │3 │11 │0 │37 │

├──────────────────────────────────────────────┼─────────────┼─────────────┼────────────────┼───────────────┤

│"Cloudy with a Chance of Meatballs (2009)" │3 │11 │0 │37 │

├──────────────────────────────────────────────┼─────────────┼─────────────┼────────────────┼───────────────┤

│"Ice Age 2: The Meltdown (2006)" │4 │6 │1 │37 │

├──────────────────────────────────────────────┼─────────────┼─────────────┼────────────────┼───────────────┤

│"22 Jump Street (2014)" │3 │8 │1 │36 │

├──────────────────────────────────────────────┼─────────────┼─────────────┼────────────────┼───────────────┤

│"Grown Ups 2 (2013)" │1 │13 │1 │36 │

├──────────────────────────────────────────────┼─────────────┼─────────────┼────────────────┼───────────────┤

│"Madagascar: Escape 2 Africa (2008)" │6 │3 │0 │36 │

├──────────────────────────────────────────────┼─────────────┼─────────────┼────────────────┼───────────────┤

│"Toy Story 3 (2010)" │3 │10 │0 │35 │

├──────────────────────────────────────────────┼─────────────┼─────────────┼────────────────┼───────────────┤

│"Despicable Me 2 (2013)" │3 │9 │0 │33 │

└──────────────────────────────────────────────┴─────────────┴─────────────┴────────────────┴───────────────┘

Started streaming 10 records after 64342 ms and completed after 64342 ms.

Here is somewhat simplified query with only actors to visualize connection:

MATCH (me:User{id:'318'})-[r:RATED]-(m:Movie)

WITH me, avg(r.rating) AS average

MATCH (me)-[r:RATED]->(m:Movie)

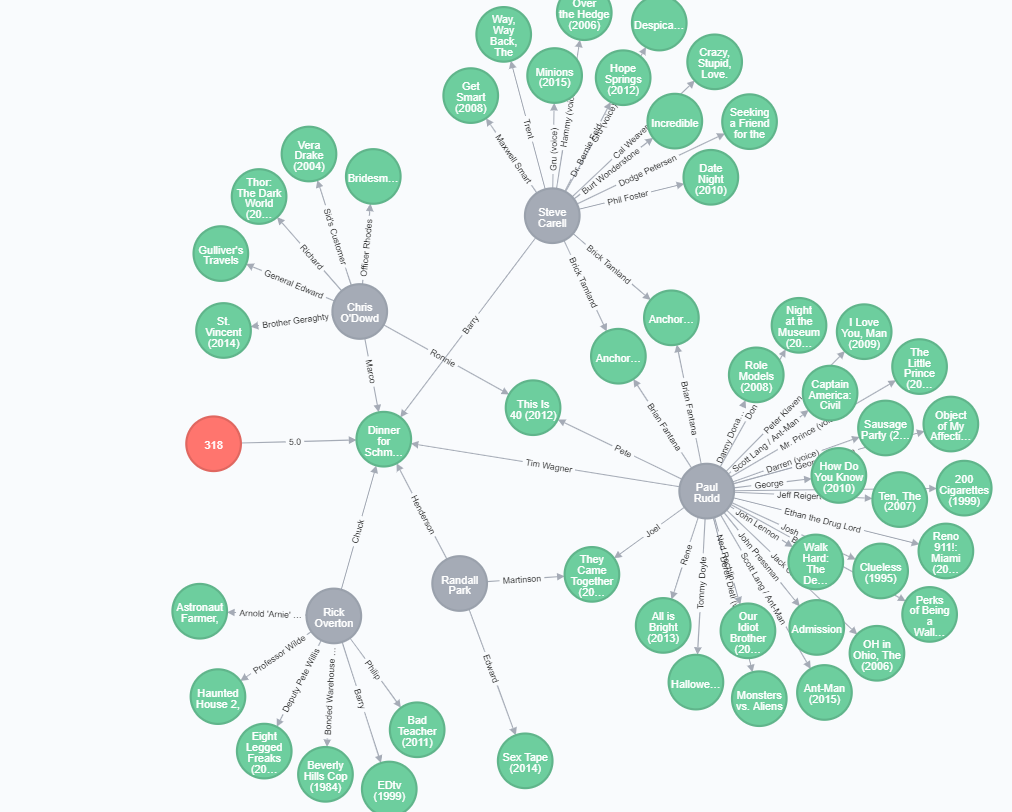
WHERE r.rating > 4.5

MATCH (m)<-[:ACTED\_IN]-(a:Person)-[:ACTED\_IN]->(rm)

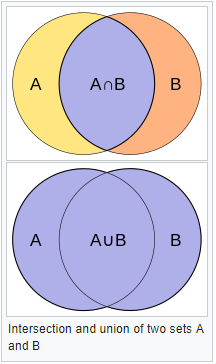
MATCH (rm)

WHERE NOT (me)-[:RATED]->(rm)

RETURN rm, a, me ,m LIMIT 50



It shows that our user liked movie Dinner for Schmucks (2010), where Paul Rudd, Rick Overton, and others played, so we’ll take a look at the movies they played at. In original query, we sorted recommendations by the number of overlapping paths that lead to a particular recommended movie.

By this moment, we used a number of paths that lead to particular movies as a score. Now let’s use Jaccard I index as a similarity metric. It is calculated as  cardinality (number of elements) of the intersection of 2 sets divided by the cardinality of the union of 2 sets:

With some help of <http://guides.neo4j.com/sandbox/recommendations> let’s show how does it work:

MATCH (me:User{id:'220'})-[r:RATED]-(m:Movie)

WITH me, avg(r.rating) AS mean

MATCH (me)-[r:RATED]->(m:Movie)

WHERE r.rating =5

MATCH (m)-[:ACTED\_IN|:DIRECTED]-(t)-[:ACTED\_IN|:DIRECTED]-(other:Movie)

WHERE NOT (me)-[:RATED]->(other)

WITH me, m, other, COUNT(t) AS intersection, COLLECT(t.name) AS i

MATCH (m)-[:ACTED\_IN|:DIRECTED]-(mt)

WITH me, m,other, intersection,i, COLLECT(mt.name) AS s1

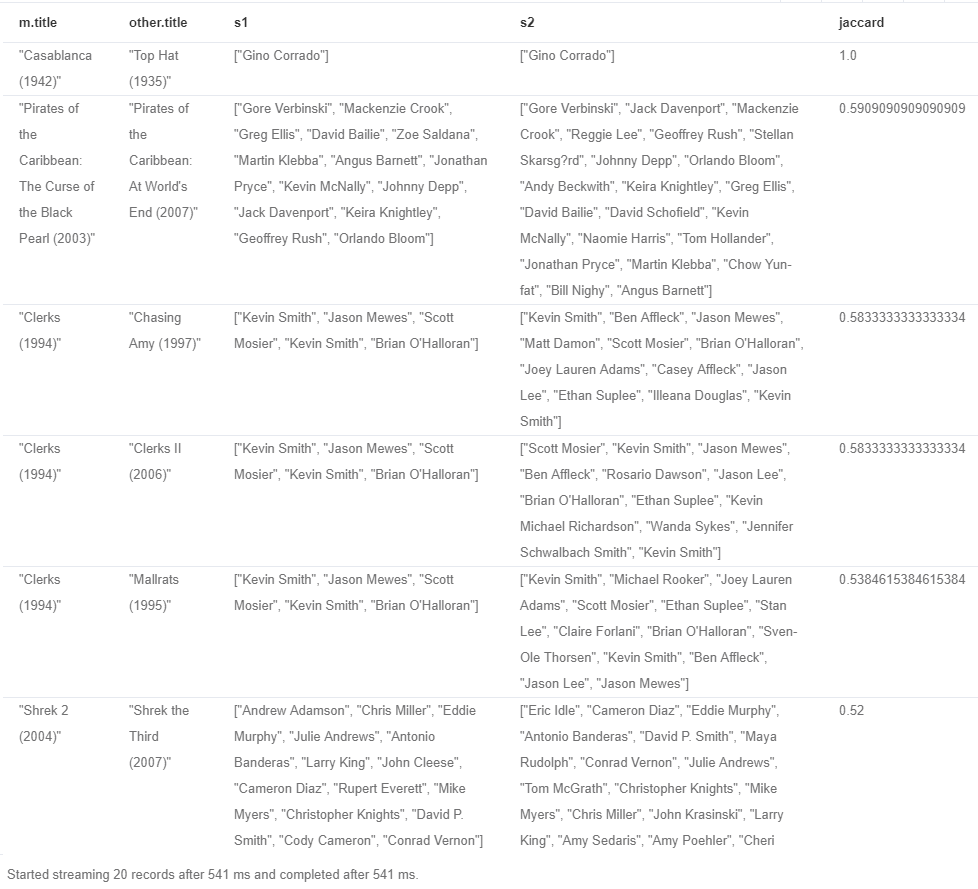
MATCH (other)-[:ACTED\_IN|:DIRECTED]-(ot)

WITH me, m,other,intersection,i, s1, COLLECT(ot.name) AS s2

WITH me, m,other,intersection,s1,s2

WITH me, m,other,intersection,s1+filter(x IN s2 WHERE NOT x IN s1) AS union, s1, s2

RETURN m.title, other.title, s1,s2,((1.0\*intersection)/SIZE(union)) AS jaccard ORDER BY jaccard DESC LIMIT 20



Except for obvious recommendations like movies from the same sequence, we ‘got pretty good math of “Clerk” and “Chasing Amy” and so forth.

Let’s go back to Collaborative Filtering. Instead of taking into consideration the opinion of all users in the system, let’s find most “similar” users; users who have the same taste. The easiest way to do so is to find the correlation coefficient between the targeted user and others, and then use ratings given only by “same minded” users.

We’ll use sample Pearson correlation coefficient, which is defined as follows:

where

is sample size;

are the individual sample points indexed with ;

- the sample mean; and analogously for .

Let’s find users with a large correlation coefficient between ratings given by our user and all others:

MATCH (me:User {id:"220"})-[r:RATED]->(m:Movie)

WITH me, avg(r.rating) AS my\_average

MATCH (me)-[r1:RATED]->(m:Movie)<-[r2:RATED]-(other)

WITH me, my\_average, other, COLLECT({r1: r1, r2: r2}) AS ratings WHERE size(ratings) > 10

MATCH (other)-[r:RATED]->(m:Movie)

WITH me, my\_average, other, avg(r.rating) AS other\_average, ratings

UNWIND ratings AS r

WITH sum( (r.r1.rating- my\_average) \* (r.r2.rating- other\_average) ) AS a,

sqrt( sum( (r.r1.rating - my\_average)^2) \* sum( (r.r2.rating - other\_average) ^2)) AS b,

me, other

WHERE b <> 0

RETURN me.id, other.id, a/b as correlation

ORDER BY correlation DESC LIMIT 10

╒═══════╤══════════╤══════════════════╕

│"me.id"│"other.id"│"correlation" │

╞═══════╪══════════╪══════════════════╡

│"220" │"494" │0.7825315077845476│

├───────┼──────────┼──────────────────┤

│"220" │"32" │0.7818916367269141│

├───────┼──────────┼──────────────────┤

│"220" │"485" │0.7633105914491696│

├───────┼──────────┼──────────────────┤

│"220" │"97" │0.7547965924537339│

├───────┼──────────┼──────────────────┤

│"220" │"79" │0.7399103445131804│

├───────┼──────────┼──────────────────┤

│"220" │"88" │0.7328107458190376│

├───────┼──────────┼──────────────────┤

│"220" │"124" │0.718593011670177 │

├───────┼──────────┼──────────────────┤

│"220" │"235" │0.7165393509283289│

├───────┼──────────┼──────────────────┤

│"220" │"436" │0.7043763345369647│

├───────┼──────────┼──────────────────┤

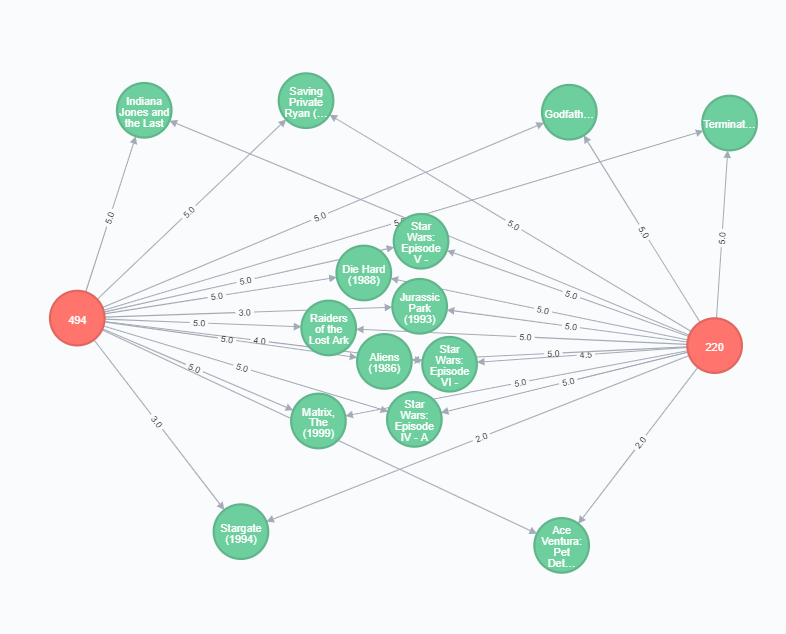
│"220" │"500" │0.6597414172261901│

├───────┼──────────┼──────────────────┤

Let’s show how does similarly rated movies looks like for targeted user and the one with largest correlation value:

MATCH (me:User {id:"220"})-[:RATED]->(m:Movie)

MATCH (other:User {id:"494"})-[:RATED]->(m:Movie)

RETURN me, other, m

As we see, highly rated movies by user 220 are also highly rated by user 494; poorly rated movies by user 220 are also poorly rated by user 494.

Let’s use this property to find recommended movies:

MATCH (me:User {id:"220"})-[r:RATED]->(m:Movie)

WITH me, avg(r.rating) AS my\_average

MATCH (me)-[r1:RATED]->(m:Movie)<-[r2:RATED]-(other)

WITH me, my\_average, other, COLLECT({r1: r1, r2: r2}) AS ratings WHERE size(ratings) > 10

MATCH (other)-[r:RATED]->(m:Movie)

WITH me, my\_average, other, avg(r.rating) AS other\_average, ratings

UNWIND ratings AS r

WITH sum( (r.r1.rating- my\_average) \* (r.r2.rating- other\_average) ) AS a,

sqrt( sum( (r.r1.rating - my\_average)^2) \* sum( (r.r2.rating - other\_average) ^2)) AS b,

me, other

WHERE b <> 0

WITH me, other, a/b as correlation

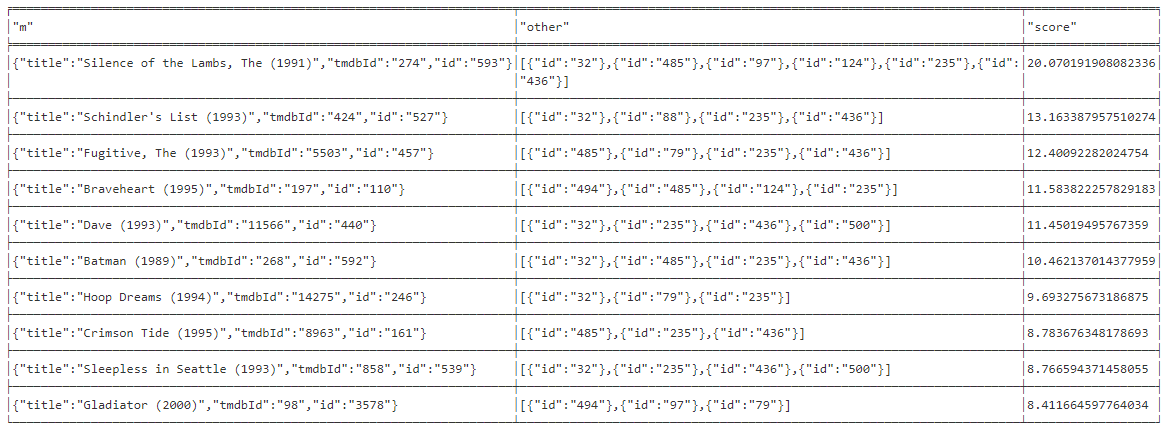
ORDER BY correlation DESC LIMIT 10

MATCH (other)-[r:RATED]->(m:Movie) WHERE NOT EXISTS( (me)-[:RATED]->(m) )

WITH m, SUM( correlation\* r.rating) AS score, COLLECT(other) AS other

RETURN m, other, score

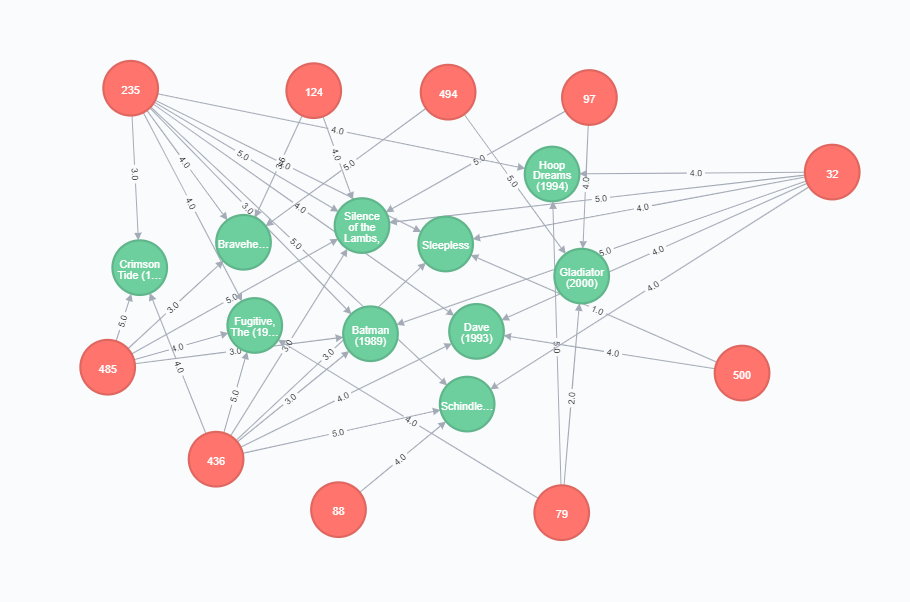
ORDER BY score DESC LIMIT 10



Started streaming 25 records after 237 ms and completed after 237 ms.

Here we see movie title, list of users, an opinion of those was taking into consideration, and score, which sum by the number of users of the rating given by user multiplied by the correlation coefficient of this user.

Here is the visualization.



We can find here user who are highly correlated with user 220, and their ratings toward chosen movies. 6 of such users gave high rate to leading movie Silence of the Lambs.

Similarly, we can find users with negative correlation: if the targeted user like particular movies, the user with high negative correlation will hate it, and opposite. Then we can use such “anti-recommendation” and hide these movies from the user in order not to upset him ☺.

MATCH (me:User {id:"220"})-[r:RATED]->(m:Movie)

WITH me, avg(r.rating) AS my\_average

MATCH (me)-[r1:RATED]->(m:Movie)<-[r2:RATED]-(other)

WITH me, my\_average, other, COLLECT({r1: r1, r2: r2}) AS ratings WHERE size(ratings) > 10

MATCH (other)-[r:RATED]->(m:Movie)

WITH me, my\_average, other, avg(r.rating) AS other\_average, ratings

UNWIND ratings AS r

WITH sum( (r.r1.rating- my\_average) \* (r.r2.rating- other\_average) ) AS a,

sqrt( sum( (r.r1.rating - my\_average)^2) \* sum( (r.r2.rating - other\_average) ^2)) AS b,

me, other

WHERE b <> 0

WITH me, other, a/b as correlation

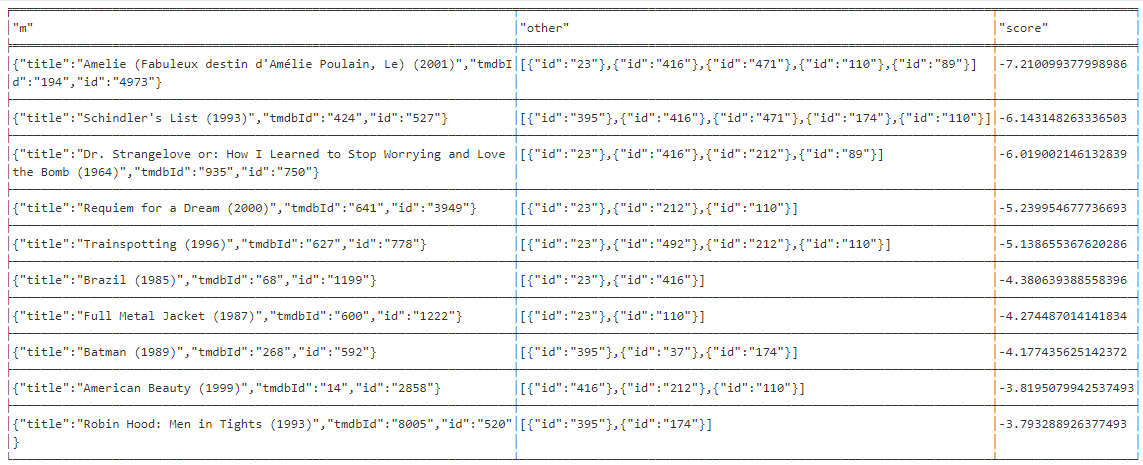
ORDER BY correlation ASC LIMIT 10

MATCH (other)-[r:RATED]->(m:Movie) WHERE NOT EXISTS( (me)-[:RATED]->(m) )

WITH m, SUM( correlation\* r.rating) AS score, COLLECT(other) AS other

RETURN m, other, score

ORDER BY score ASC LIMIT 10



# Conclusions

I used neo4j graph database and declarative graph query language Cypher to create a model for movie recommendation system using previous user experience. As a data source, I choose 2 separate databases – MovieLens, which contains ratings and tag applications applied to movies by users and TMDB 5000 Movie Dataset, which gave me access to movies actors, directors. Data from 2 datasets were united using links.csv file which contains both “internal” movie id (used thought MovieLens files) and “foreign” id which refers to movie id in TMDB 5000 Movie Dataset.

Neo4j fits perfectly for this task. We constantly have to use connections between entities, like find movies likes by user1 which also are liked by other users, and then find movies that other users liked, but user1 hasn’t seen. Had we user traditional relational database, we’d end up with a large number of joints, which are very expensive for RDBMS. With a graph database, on the other hand, we have fast access to both data (user, movie, genre) and relationships between them. As all relationships are easily and quickly acceptable, it allows us to process queries very fast, enabling using the model for real-time recommendation engines.

Most queries used in this work took about 200-500 ms to process. The longest query took ~60000 ms, in RDBMS it would require ~10 joints and would take much longer.

Another advantage of using a graph database for this model is that it’s easy to visualize the connections and paths that led us to a particular result, and by doing so, to understand the underlying patter better.

Graph query language Cypher is very easy to learn but very powerful. It allows a user to write moderately complex queries even without prior knowledge of this language. I, for example, have never used it before, except during one homework in this course, yet, I thoroughly enjoyed working with it.

I used different models – both Content-Based, Collaborative Filtering and combination of them. It’s hard to evaluate the performance of such models. We would have to propose movies to a user, and then to see whether he or she liked them. We would need “access” to a real user to do so.

It would be interesting to use other features to expand our model, like user demographic information, social relationships; more consistent tags that describe the movies, as well as more information about movies itself, like to know the movie sequences (we wouldn’t want to recommend user to watch episode #8 long sequence, if he had never watched any previous, even if his friends like it, rather, it would be better to recommend him to watch from the beginning).

As I discovered, the problem of creating a model for a recommendation engine, in particular, for movies recommendation system, can be successfully and easily solved using a graph database.

# References:

Code and technical info:

<https://neo4j.com/>

<https://anaconda.org/anaconda/anaconda-navigator>

<https://www.python.org/>

<http://jupyter.org/>

<http://guides.neo4j.com/sandbox/recommendations>

<https://neo4j.com/developer/movie-database/#_import_instructions>

<https://neo4j.com/graphgist/competency-management-a-matter-of-filtering-and-recommendation-engines#competences>

<https://github.com/citruz/movies4j>

<https://neo4j.com/blog/real-time-recommendation-engine-data-science/>

<https://en.wikipedia.org/wiki/Jaccard_index>

<https://en.wikipedia.org/wiki/Pearson_correlation_coefficient>

Data source:

<https://www.kaggle.com/tmdb/tmdb-movie-metadata>

<https://www.themoviedb.org/>

<https://grouplens.org/datasets/movielens/>

F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. ACM Transactions on Interactive Intelligent Systems (TiiS) 5, 4: 19:1–19:19. <https://doi.org/10.1145/2827872>