Politechnika Świętokrzyska
Kielce University of Technology
Келецкий технический университет

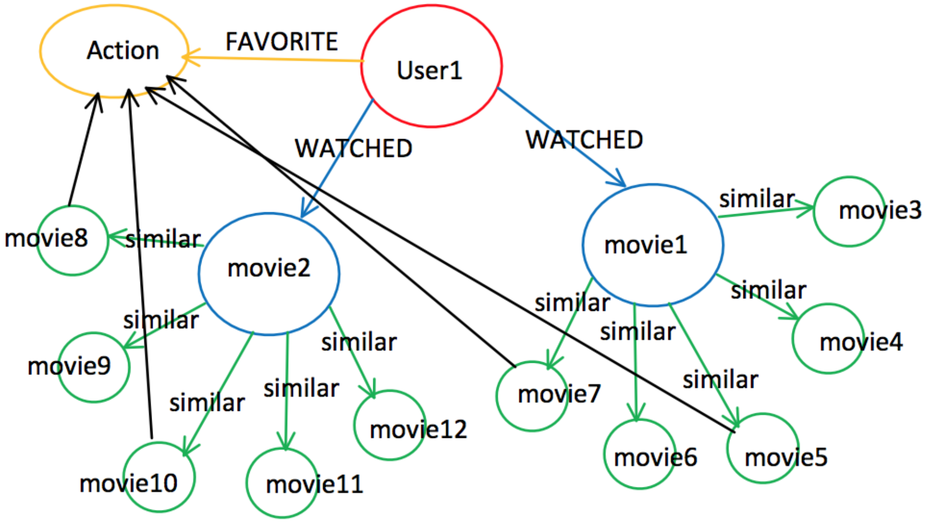
Movie Recommendation System using Neo4j

Project | Databases II

Mgr inż. Łukasz Zawarczyński
Tanmay Nandanikar

1. Introduction

Shown below is the basic structure of the database:



A movie recommendation system using Neo4j and python can create a list of movies that the user might actually like.

Neo4j is a graph database that can be used to store and query data. In this project, we will use Neo4j to store information about movies and connect movies with each other using their similarity score. We will also use python to create a script that will recommend movies to the user.

2. The dataset

We will be using the following dataset:

https://files.grouplens.org/datasets/movielens/ml-25m.zip

First, we will use the following code to limit the dataset to the first 1000 movies:

```
import pandas as pd

data = pd.read_csv('ratings.csv')
df = pd.DataFrame(columns=[ 'userID', 'movieID', 'rating', 'timestamp' ])

# fields = ['userID', 'movieID', 'rating', 'timestamp']
new_ratings = [ ]
print("done0")
print(data.head(1))
print("done1")
m = 0
```

```
k = 100
for i, j in data.iterrows():
    if j[ 1 ] < 1001:
        m += 1
        df.loc[ m ] = [ j[ 0 ], j[ 1 ], j[ 2 ], j[ 3 ] ]
        if (k < m):
            print(m)
            k += 10000

df.to_csv(test.csv, encoding='utf-8', index=False)
```

We can do the same for the files ‘genome-scores.csv’ and ‘movies.csv’.

Next, we import the data to python using the following code:

```
import pandas as pd
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
from collections import Counter

genome_scores_data = pd.read_csv('genome-scores.csv')
movies_data = pd.read_csv('movies.csv')
ratings_data = pd.read_csv('ratings.csv')

movies_df = movies_data.drop(['genres','movieId'], axis = 1)
movies_df.to_csv('movies1.csv', sep='|', header=True, index=False)
```

MOVIE SIMILARITY DATASET

We are going to create a movie similarity matrix using cosine similarity. The matrix is going to be created by mixing cosine similarities of movie tags, movie genres, and movie ratings. The movie_tag dataframe is created using the genome scores by merging it with the movie dataset using movie_id as key.

```
scores_pivot = genome_scores_data.pivot_table(index = ["movieId"],columns = ["tagId"],values
= "relevance").reset_index()
mov_tag_df = movies_data.merge(scores_pivot, left_on='movieId', right_on='movieId',
how='left')
mov_tag_df = mov_tag_df.fillna(0)
mov_tag_df = mov_tag_df.drop(['title','genres'], axis = 1)
```

The movie genre dataset is generated by splitting the genre column from the movie dataframe into individual columns for each genre. the value of each cell is set to 1 if the movie belongs to that genre and otherwise.

```
def set_genres(genres,col):
    if genres in col.split('|'): return 1
    else: return 0


mov_genres_df = movies_data.copy()
mov_genres_df["Action"] = mov_genres_df.apply(lambda x: set_genres("Action",x['genres']),
axis=1)
mov_genres_df["Adventure"] = mov_genres_df.apply(lambda x:
set_genres("Adventure",x['genres']), axis=1)
```

```
mov_genres_df["Animation"] = mov_genres_df.apply(lambda x:
set_genres("Animation",x['genres']), axis=1)
mov_genres_df["Children"] = mov_genres_df.apply(lambda x:
set_genres("Children",x['genres']), axis=1)
mov_genres_df["Comedy"] = mov_genres_df.apply(lambda x: set_genres("Comedy",x['genres']),
axis=1)
mov_genres_df["Crime"] = mov_genres_df.apply(lambda x: set_genres("Crime",x['genres']),
axis=1)
mov_genres_df["Documentary"] = mov_genres_df.apply(lambda x:
set_genres("Documentary",x['genres']), axis=1)
mov_genres_df["Drama"] = mov_genres_df.apply(lambda x: set_genres("Drama",x['genres']),
axis=1)
mov_genres_df["Fantasy"] = mov_genres_df.apply(lambda x: set_genres("Fantasy",x['genres']),
axis=1)
mov_genres_df["Film-Noir"] = mov_genres_df.apply(lambda x: set_genres("Film-
Noir",x['genres']), axis=1)
mov_genres_df["Horror"] = mov_genres_df.apply(lambda x: set_genres("Horror",x['genres']),
axis=1)
mov_genres_df["Musical"] = mov_genres_df.apply(lambda x: set_genres("Musical",x['genres']),
axis=1)
mov_genres_df["Mystery"] = mov_genres_df.apply(lambda x: set_genres("Mystery",x['genres']),
axis=1)
mov_genres_df["Romance"] = mov_genres_df.apply(lambda x: set_genres("Romance",x['genres']),
axis=1)
mov_genres_df["Sci-Fi"] = mov_genres_df.apply(lambda x: set_genres("Sci-Fi",x['genres']),
axis=1)
mov_genres_df["Thriller"] = mov_genres_df.apply(lambda x:
set_genres("Thriller",x['genres']), axis=1)
mov_genres_df["War"] = mov_genres_df.apply(lambda x: set_genres("War",x['genres']), axis=1)
mov_genres_df["Western"] = mov_genres_df.apply(lambda x: set_genres("Western",x['genres']),
axis=1)
mov_genres_df["(no genres listed)"] = mov_genres_df.apply(lambda x: set_genres("(no genres
listed)",x['genres']), axis=1)
mov_genres_df.drop(['title','genres'], axis = 1, inplace=True)
```

MOVIE RATING DATASET

The movie rating dataset is created using movies and rating datasets. it contains the rating, rating count, and year group of the movie. The year group has values from 0–5 and the rating count has values from 0–5.

```
def set_year(title):
    year = title.strip()[-5:-1]
    if year.isnumeric():
        return int(year)
    else:
        return 1800


def set_year_group(year):
    if (year < 1900): return 0
    elif (1900 <= year <= 1975): return 1
    elif (1976 <= year <= 1995): return 2
    elif (1996 <= year <= 2003): return 3
    elif (2004 <= year <= 2009): return 4
    elif (2010 <= year): return 5
    else: return 0


def set_rating_group(rating_counts):
    if (rating_counts <= 1): return 0
    elif (2 <= rating_counts <= 10): return 1
    elif (11 <= rating_counts <= 100): return 2
    elif (101 <= rating_counts <= 1000): return 3
    elif (1001 <= rating_counts <= 5000): return 4
    elif (5001 <= rating_counts): return 5
    else: return 0


movies = movies_data.copy()
movies['year'] = movies.apply(lambda x: set_year(x['title']), axis=1)
movies['year_group'] = movies.apply(lambda x: set_year_group(x['year']), axis=1)
movies.drop(['title','year'], axis = 1, inplace=True)


agg_movies_rat = ratings_data.groupby(['movieId']).agg({'rating': [np.size,
np.mean]}).reset_index()
agg_movies_rat.columns = ['movieId','rating_counts', 'rating_mean']
agg_movies_rat['rating_group'] = agg_movies_rat.apply(lambda x:
set_rating_group(x['rating_counts']), axis=1)
agg_movies_rat.drop('rating_counts', axis = 1, inplace=True)
mov_rating_df = movies.merge(agg_movies_rat, left_on='movieId', right_on='movieId',
how='left')
mov_rating_df = mov_rating_df.fillna(0)
mov_rating_df.drop(['genres'], axis = 1, inplace=True)
```

The final cosine similarly score is calculated by adding the similarity scores of movie tags, movie genres, and movie ratings with the weights of 0.5,0.25 and 0.25 respectively

```
mov_tag_df = mov_tag_df.set_index('movieId')
mov_genres_df = mov_genres_df.set_index('movieId')
mov_rating_df = mov_rating_df.set_index('movieId')


cos_tag = cosine_similarity(mov_tag_df.values)*0.5
cos_genres = cosine_similarity(mov_genres_df.values)*0.25
cos_rating = cosine_similarity(mov_rating_df.values)*0.25
cos = cos_tag+cos_genres+cos_rating


cols = mov_tag_df.index.values
inx = mov_tag_df.index
movies_sim = pd.DataFrame(cos, columns=cols, index=inx)
movies_sim.head()
```

PREPARE OTHER DATASETS

First we write the function to find movies similar to a given movie.

We create “users”, “movies”, “genres”, “users_movies”, “movies_genres”, “users_genres” datasets. We use “users”, “movies” and “genres” datasets for nodes and others for relationships.

```
def get_similar(movieId):
    df = movies_sim.loc[movies_sim.index == movieId].reset_index(). \
            melt(id_vars='movieId', var_name='sim_moveId', value_name='relevance'). \
            sort_values('relevance', axis=0, ascending=False)[1:6]
    return df
```

```
movies_similarity = pd.DataFrame(columns=['movieId','sim_moveId','relevance'])
for x in movies_sim.index.tolist():
    movies_similarity = movies_similarity.append(get_similar(x))


print(movies_similarity.head())

users_df = pd.DataFrame(ratings_data['userId'].unique(), columns=['userId'])

#create movies_df
movies_df = movies_data.drop('genres', axis = 1)
#calculate mean of ratings for each movies
agg_rating_avg = ratings_data.groupby(['movieId']).agg({'rating': np.mean}).reset_index()
agg_rating_avg.columns = ['movieId', 'rating_mean']
#merge
movies_df = movies_df.merge(agg_rating_avg, left_on='movieId', right_on='movieId',
how='left')

genres = [
    "Action",
    "Adventure",
    "Animation",
    "Children",
    "Comedy",
    "Crime",
    "Documentary",
    "Drama",
    "Fantasy",
    "Film-Noir",
    "Horror",
    "Musical",
    "Mystery",
    "Romance",
    "Sci-Fi",
    "Thriller",
    "War",
    "Western",
    "(no genres listed)"]
genres_df = pd.DataFrame(genres, columns=['genres'])

users_movies_df = ratings_data.drop('timestamp', axis = 1)

movies_genres_df = movies_data.drop('title', axis = 1)


#define a function to split genres field
def get_movie_genres(movieId):
    movie = movies_genres_df[movies_genres_df['movieId']==movieId]
    genres = movie['genres'].tolist()
    df = pd.DataFrame([b for a in [i.split('|') for i in genres] for b in a],
columns=['genres'])
    df.insert(loc=0, column='movieId', value=movieId)
    return df

#create empty df
movies_genres=pd.DataFrame(columns=['movieId','genres'])
for x in movies_genres_df['movieId'].tolist():
    movies_genres=movies_genres.append(get_movie_genres(x))


#join to movies data to get genre information
user_genres_df = ratings_data.merge(movies_data, left_on='movieId', right_on='movieId',
how='left')
```

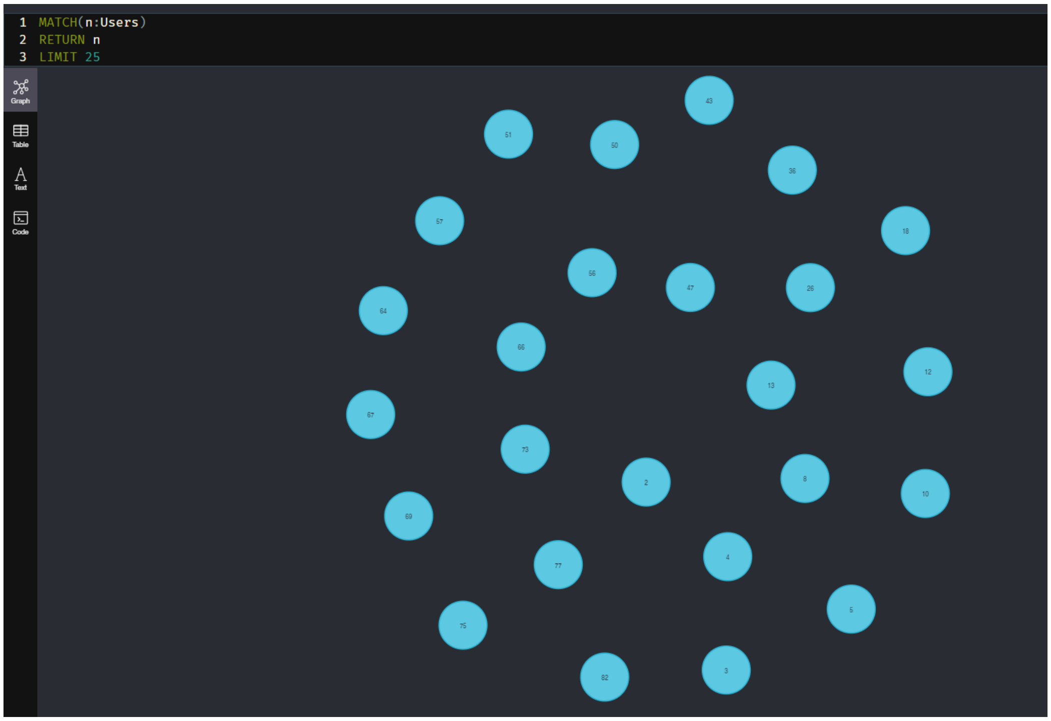
```
#drop columns that will not be used
user_genres_df.drop(['movieId','rating','timestamp','title'], axis = 1, inplace=True)

def get_favorite_genre(userId):
    user = user_genres_df[user_genres_df['userId']==userId]
    genres = user['genres'].tolist()
    movie_list = [b for a in [i.split('|') for i in genres] for b in a]
    counter = Counter(movie_list)
    return counter.most_common(1)[ 0 ][ 0 ]

#create empty df
users_genres = pd.DataFrame(columns=['userId','genre'])
for x in users_df['userId'].tolist():
    users_genres = users_genres.append(pd.DataFrame([[x,get_favorite_genre(x)]],
columns=['userId','genre']))
```

We create 7 datasets so we are ready to create recommendation database in neo4j. First let's export our datasets as csv files.

```
users_df.to_csv('users1.csv', sep='|', header=True, index=False)
movies_df.to_csv('movies2.csv', sep='|', header=True, index=False)
genres_df.to_csv('genres1.csv', sep='|', header=True, index=False)
users_movies_df.to_csv('users_movies1.csv', sep='|', header=True, index=False)
movies_genres.to_csv('movies_genres1.csv', sep='|', header=True, index=False)
users_genres.to_csv('users_genres1.csv', sep='|', header=True, index=False)
movies_similarity.to_csv('movies_similarity1.csv', sep='|', header=True, index=False)
```

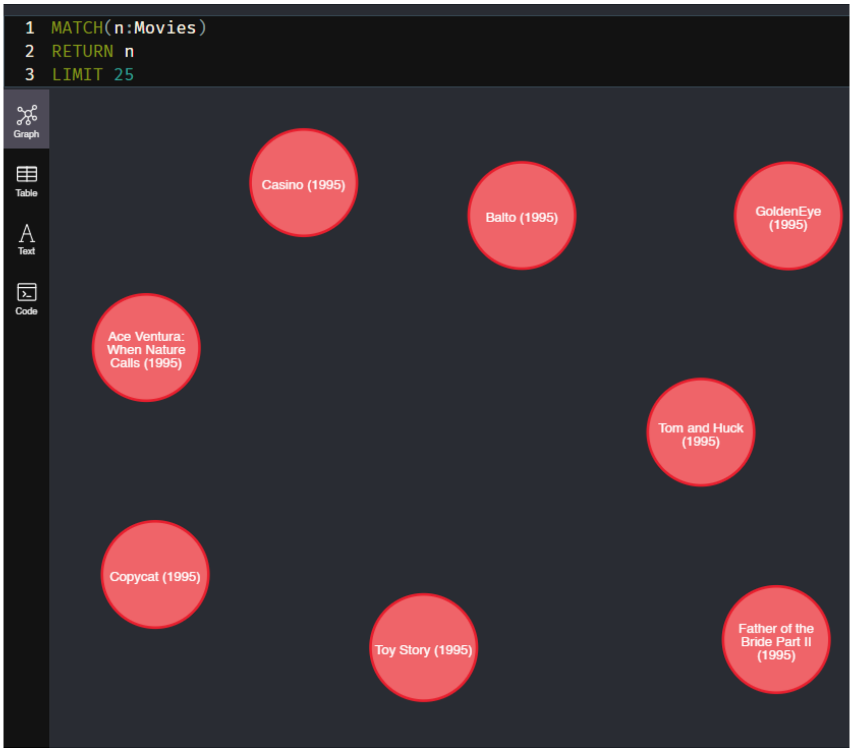
LOADING DATA TO NEO4J

```
USING PERIODIC COMMIT LOAD CSV WITH HEADERS FROM "file:///users.csv" AS row
FIELDTERMINATOR '|'
CREATE (:Users {userId: row.userId});
```



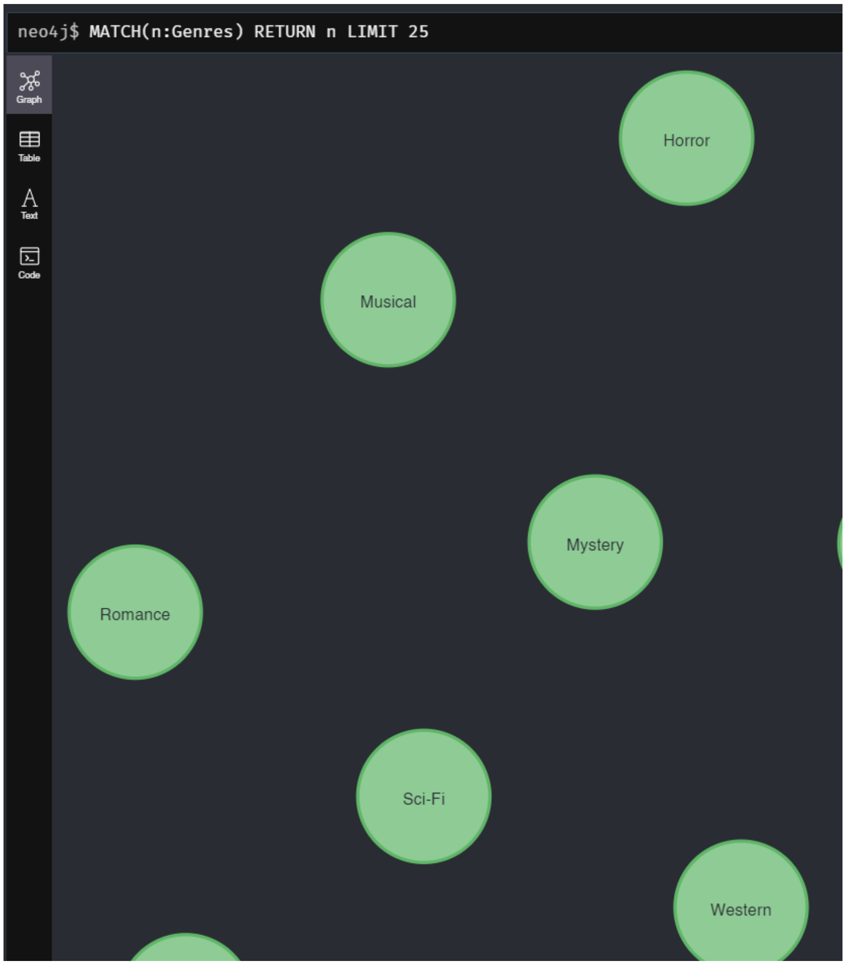
```
MATCH(n:Users)
RETURN n
LIMIT 25
```

```
USING PERIODIC COMMIT LOAD CSV WITH HEADERS FROM “file:///movies.csv” AS row
FIELDTERMINATOR ‘|’
CREATE (:Movies {movieId: row.movieId, title: row.title, rating_mean: row.rating_mean});
```



```
MATCH(n:Movies)
RETURN n
LIMIT 25
```

```
USING PERIODIC COMMIT LOAD CSV WITH HEADERS FROM "file:///genres.csv" AS row
FIELDTERMINATOR '|'
CREATE (:Genres {genres: row.genres});
```

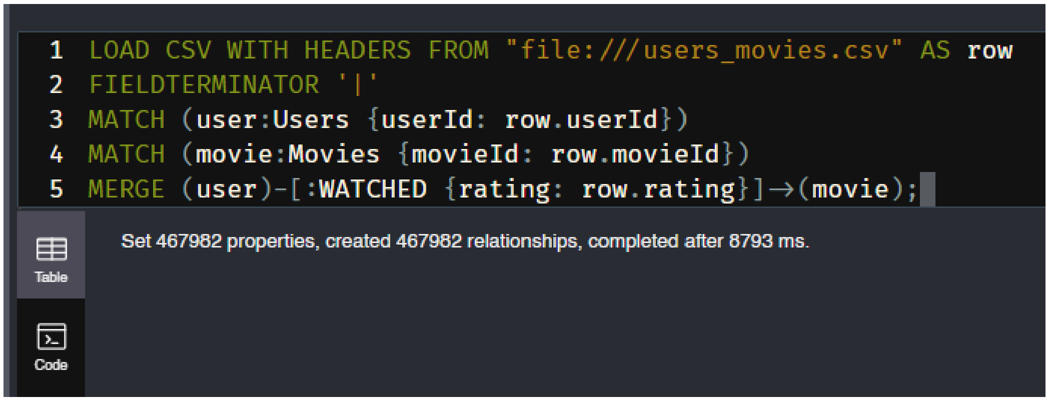


```
neo4j$ MATCH(n:Genres) RETURN n LIMIT 25
```

```
CREATE INDEX ON :Users(userId);
CREATE INDEX ON :Movies(movieId);
```

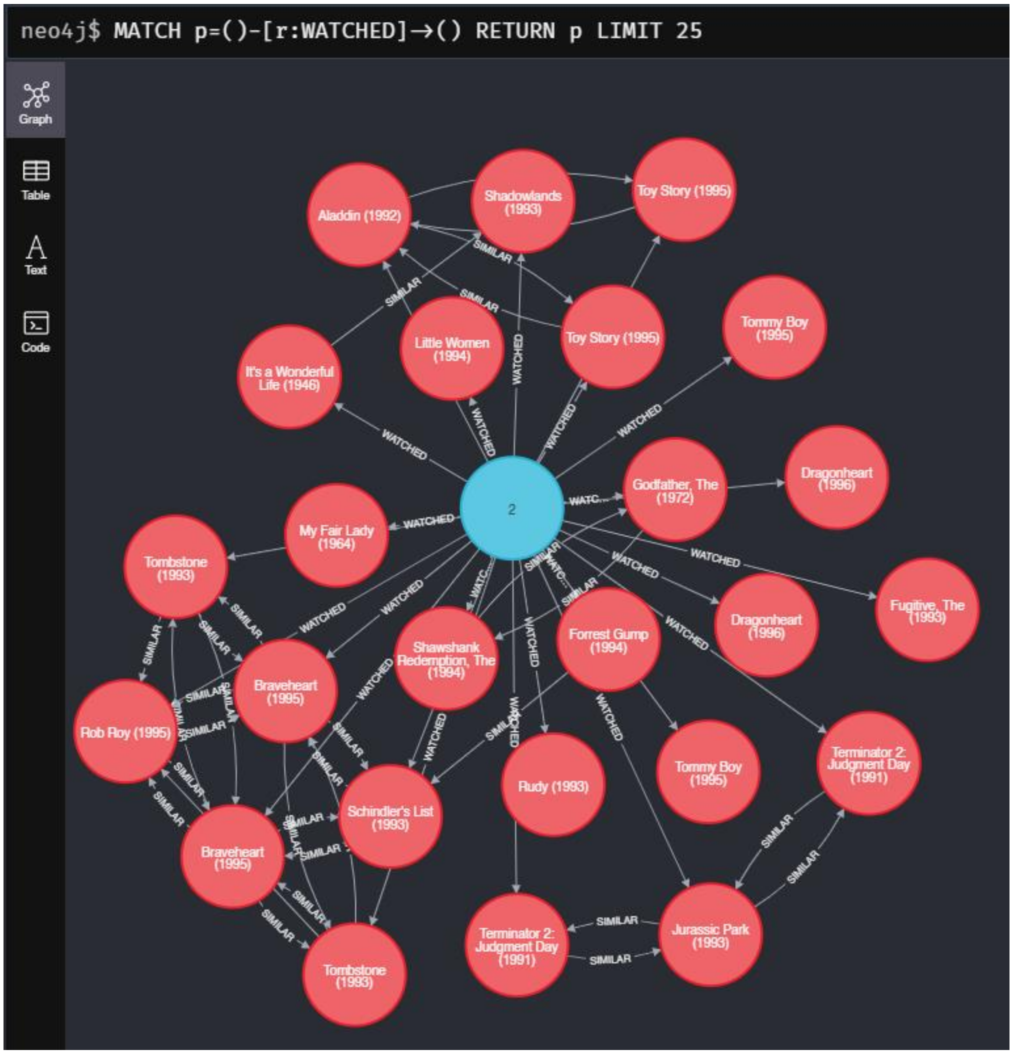
We have 4 relationships:

```
USING PERIODIC COMMIT LOAD CSV WITH HEADERS FROM "file:///users_movies.csv" AS row
FIELDTERMINATOR '|'
MATCH (user:Users {userId: row.userId})
MATCH (movie:Movies {movieId: row.movieId})
MERGE (user)-[:WATCHED {rating: row.rating}]->(movie);
```



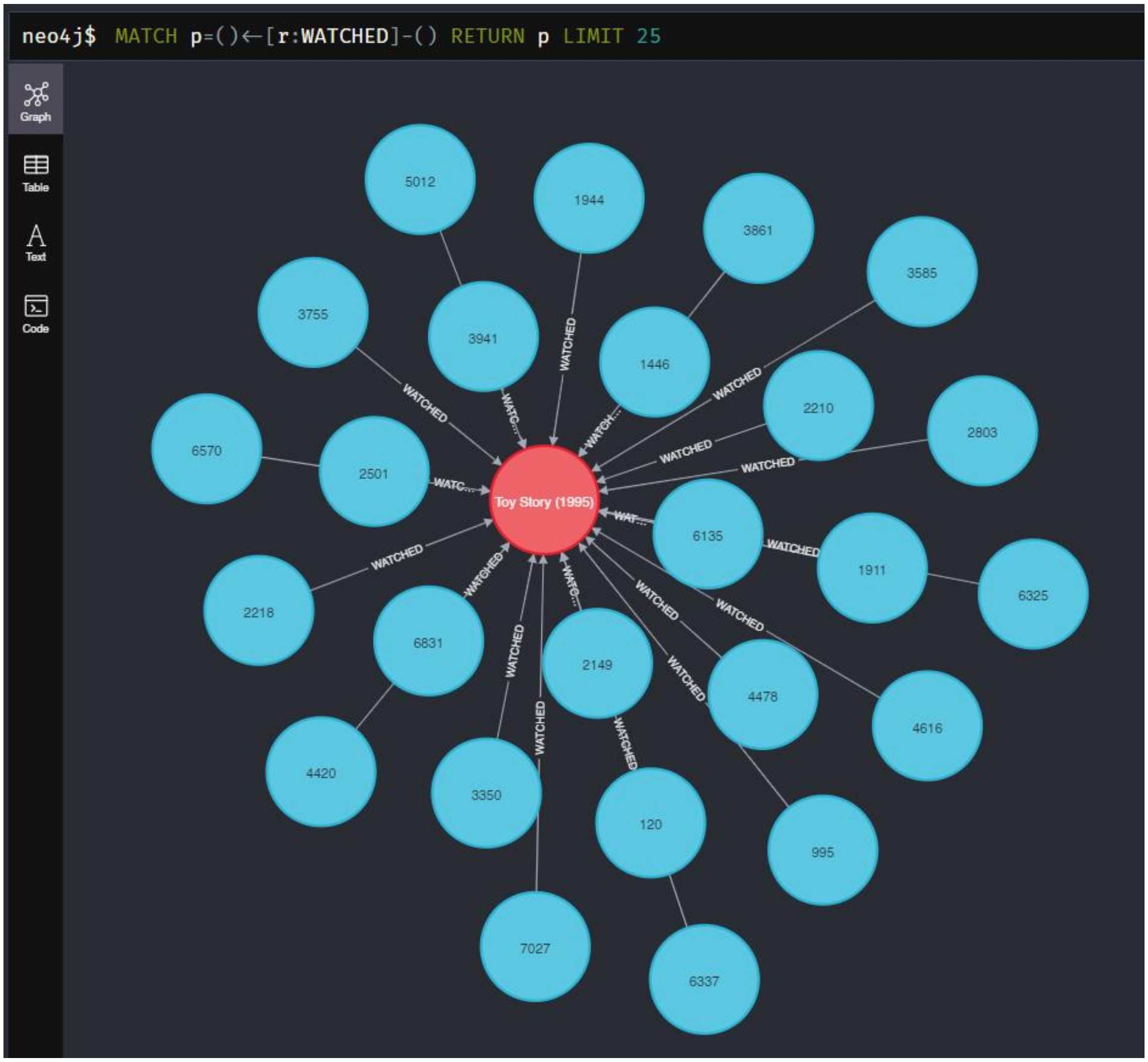
```
LOAD CSV WITH HEADERS FROM "file:///users_movies.csv" AS row
FIELDTERMINATOR '|'
MATCH (user:Users {userId: row.userId})
MATCH (movie:Movies {movieId: row.movieId})
MERGE (user)-[:WATCHED {rating: row.rating}]→(movie);
```

Set 467982 properties, created 467982 relationships, completed after 8793 ms.



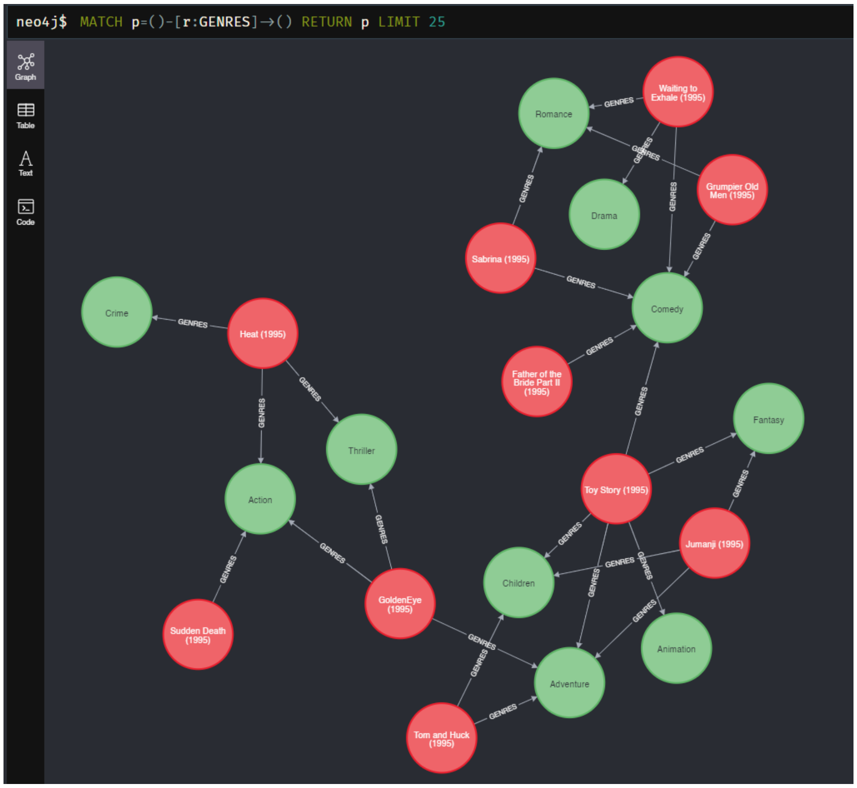
```
neo4j$ MATCH p=()-[r:WATCHED]→() RETURN p LIMIT 25
```

```
USING PERIODIC COMMIT LOAD CSV WITH HEADERS FROM “file:///users_genres.csv” AS row
FIELDTERMINATOR ‘|’
MATCH (user:Users {userId: row.userId})
MATCH (genres:Genres {genres: row.genre})
MERGE (user)-[:FAVORITE]->(genres);
```



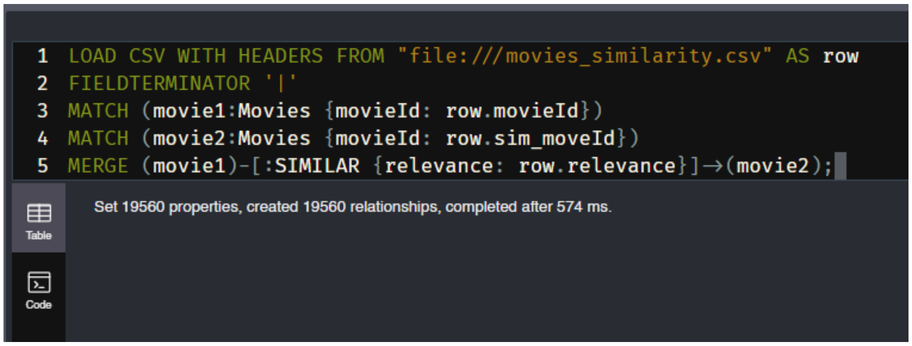
```
neo4j$ MATCH p=()←[r:WATCHED]-() RETURN p LIMIT 25
```

```
USING PERIODIC COMMIT LOAD CSV WITH HEADERS FROM “file:///movies_genres.csv” AS row
FIELDTERMINATOR ‘|’
MATCH (movie:Movies {movieId: row.movieId})
MATCH (genres:Genres {genres: row.genres})
MERGE (movie)-[:GENRES]->(genres);
```



```
neo4j$ MATCH p=()-[r:GENRES]→() RETURN p LIMIT 25
```

```
USING PERIODIC COMMIT LOAD CSV WITH HEADERS FROM “file:///movies_similarity.csv” AS row
FIELDTERMINATOR ‘|’
MATCH (movie1:Movies {movieId: row.movieId})
MATCH (movie2:Movies {movieId: row.sim_moveId})
MERGE (movie1)-[:SIMILAR {relevance: row.relevance}]->(movie2);
```



```
LOAD CSV WITH HEADERS FROM "file:///movies_similarity.csv" AS row
FIELDTERMINATOR '|'
MATCH (movie1:Movies {movieId: row.movieId})
MATCH (movie2:Movies {movieId: row.sim_moveId})
MERGE (movie1)-[:SIMILAR {relevance: row.relevance}]→(movie2);
```

Set 19560 properties, created 19560 relationships, completed after 574 ms.

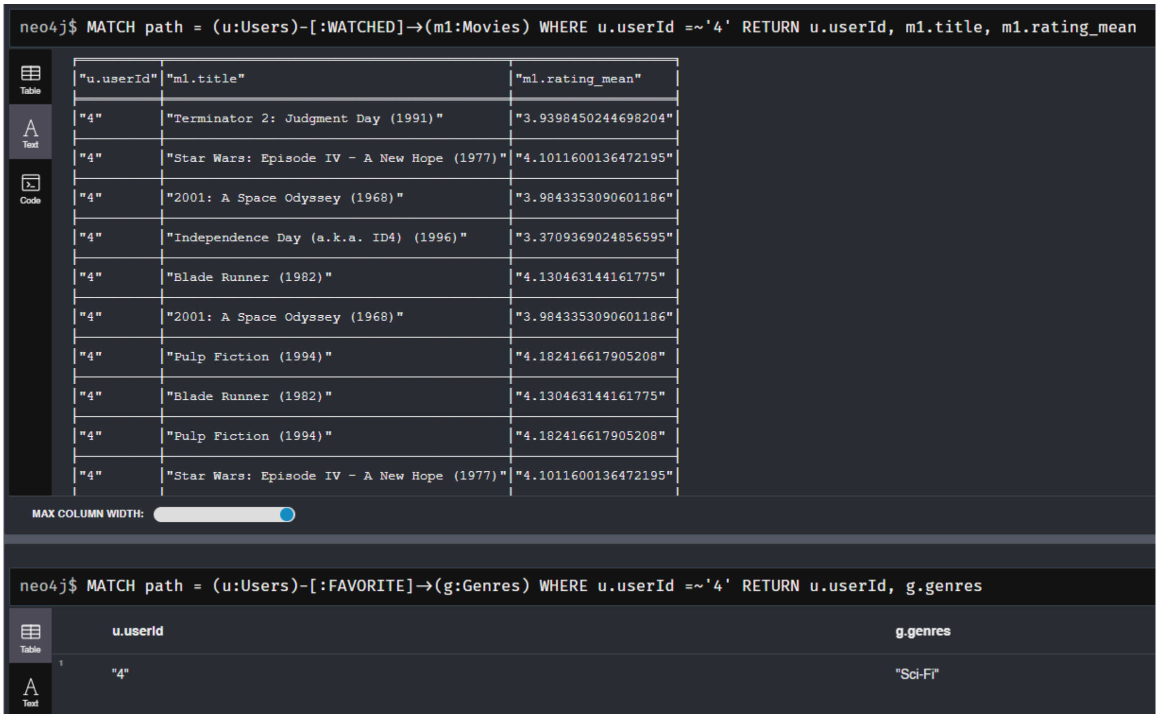
We have created the recommendation database so we can start to work on it.

QUERIES

From the following queries, we found that user with id=4 watched 14 movies and his favorite genre is Sci-Fi:

```
MATCH path = (u:Users)-[:FAVORITE]->(g:Genres)
WHERE u.userId =~'4'
RETURN u.userId, g.genres
```

```
MATCH path = (u:Users)-[:WATCHED]->(m1:Movies)
WHERE u.userId =~'4'
RETURN u.userId, m1.title, m1.rating_mean
```

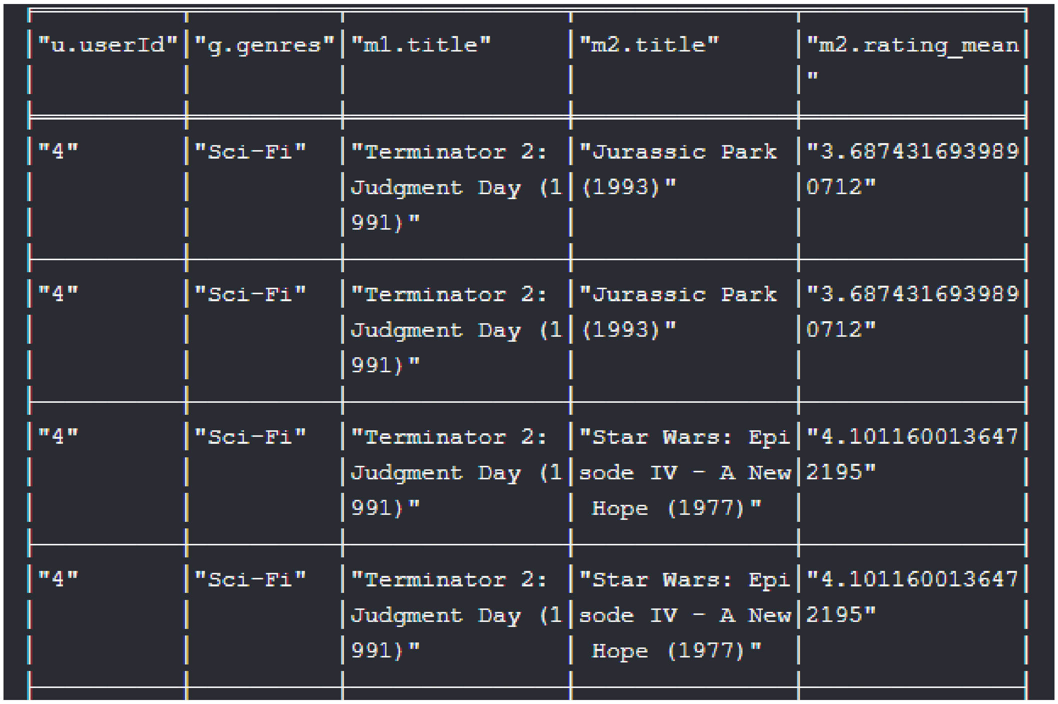


```
neo4j$ MATCH path = (u:Users)-[:WATCHED]→(m1:Movies) WHERE u.userId =~'4' RETURN u.userId, m1.title, m1.rating_mean

|"u.userId"|"m1.title"                                 |"m1.rating_mean"   |
|"4"       |"Terminator 2: Judgment Day (1991)"        |"3.9398450244698204"|
|"4"       |"Star Wars: Episode IV - A New Hope (1977)"|"4.1011600136472195"|
|"4"       |"2001: A Space Odyssey (1968)"             |"3.9843353090601186"|
|"4"       |"Independence Day (a.k.a. ID4) (1996)"     |"3.3709369024856595"|
|"4"       |"Blade Runner (1982)"                      |"4.130463144161775" |
|"4"       |"2001: A Space Odyssey (1968)"             |"3.9843353090601186"|
|"4"       |"Pulp Fiction (1994)"                      |"4.182416617905208" |
|"4"       |"Blade Runner (1982)"                      |"4.130463144161775" |
|"4"       |"Pulp Fiction (1994)"                      |"4.182416617905208" |
|"4"       |"Star Wars: Episode IV - A New Hope (1977)"|"4.1011600136472195"|
```

```
neo4j$ MATCH path = (u:Users)-[:FAVORITE]→(g:Genres) WHERE u.userId =~'4' RETURN u.userId, g.genres

    u.userId                                    g.genres
1   "4"                                         "Sci-Fi"
```

Let’s run the query below to find similar movies to the ones the user has watched, in his favorite genre (Sci-Fi):

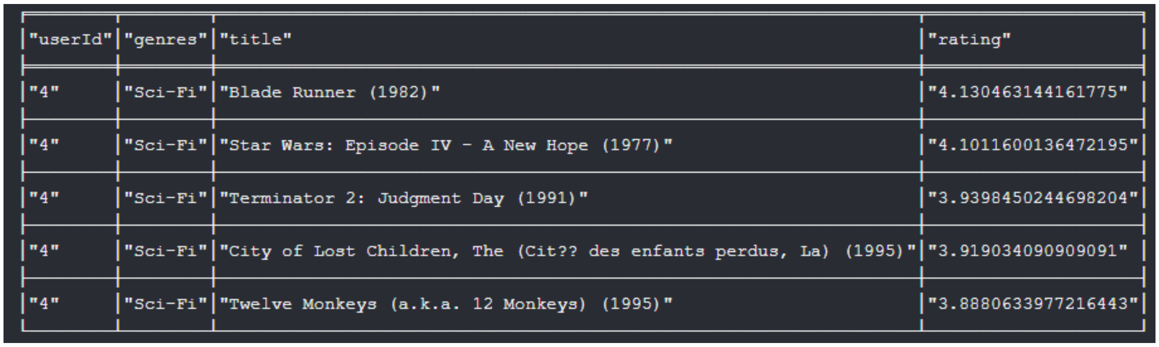
```
MATCH path = (u:Users)-[:WATCHED]->(m1:Movies)-[s:SIMILAR]->(m2:Movies),
(m2)-[:GENRES]->(g:Genres),
(u)-[:FAVORITE]->(g)
WHERE u.userId =~'4'
RETURN u.userId, g.genres, m1.title, m2.title, m2.rating_mean
```



```
╒═══════════╤═══════════╤══════════════════╤══════════════════╤═══════════════╕
│"u.userId" │"g.genres" │"m1.title"        │"m2.title"        │"m2.rating_mean│
│           │           │                  │                  │"              │
╞═══════════╪═══════════╪══════════════════╪══════════════════╪═══════════════╡
│"4"        │"Sci-Fi"   │"Terminator 2:    │"Jurassic Park    │"3.687431693989│
│           │           │Judgment Day (1   │(1993)"           │0712"          │
│           │           │991)"             │                  │               │
├───────────┼───────────┼──────────────────┼──────────────────┼───────────────┤
│"4"        │"Sci-Fi"   │"Terminator 2:    │"Jurassic Park    │"3.687431693989│
│           │           │Judgment Day (1   │(1993)"           │0712"          │
│           │           │991)"             │                  │               │
├───────────┼───────────┼──────────────────┼──────────────────┼───────────────┤
│"4"        │"Sci-Fi"   │"Terminator 2:    │"Star Wars: Epi   │"4.101160013647│
│           │           │Judgment Day (1   │sode IV - A New   │2195"          │
│           │           │991)"             │ Hope (1977)"     │               │
├───────────┼───────────┼──────────────────┼──────────────────┼───────────────┤
│"4"        │"Sci-Fi"   │"Terminator 2:    │"Star Wars: Epi   │"4.101160013647│
│           │           │Judgment Day (1   │sode IV - A New   │2195"          │
│           │           │991)"             │ Hope (1977)"     │               │
├───────────┼───────────┼──────────────────┼──────────────────┼───────────────┤
```

We can then filter movies which the user has already watched, sort them by ratings, and return only the top 5 movies based on ratings:

```
MATCH (u1:Users)-[:WATCHED]->(m3:Movies)
WHERE u1.userId =~'4'
WITH [i in m3.movieId | i] as movies
MATCH path = (u:Users)-[:WATCHED]->(m1:Movies)-[s:SIMILAR]->(m2:Movies),
(m2)-[:GENRES]->(g:Genres),
(u)-[:FAVORITE]->(g)
WHERE u.userId =~'4' and not m2.movieId in movies
RETURN distinct u.userId as userId, g.genres as genres,
m2.title as title, m2.rating_mean as rating
ORDER BY m2.rating_mean descending
LIMIT 5
```



```
╒════════╤════════╤═════════════════════════════════════════════════════════════╤════════════════════╕
│"userId"│"genres"│"title"                                                      │"rating"            │
╞════════╪════════╪═════════════════════════════════════════════════════════════╪════════════════════╡
│"4"     │"Sci-Fi"│"Blade Runner (1982)"                                        │"4.130463144161775" │
├────────┼────────┼─────────────────────────────────────────────────────────────┼────────────────────┤
│"4"     │"Sci-Fi"│"Star Wars: Episode IV - A New Hope (1977)"                  │"4.1011600136472195"│
├────────┼────────┼─────────────────────────────────────────────────────────────┼────────────────────┤
│"4"     │"Sci-Fi"│"Terminator 2: Judgment Day (1991)"                          │"3.9398450244698204"│
├────────┼────────┼─────────────────────────────────────────────────────────────┼────────────────────┤
│"4"     │"Sci-Fi"│"City of Lost Children, The (Cit?? des enfants perdus, La) (1995)"│"3.919034090909091" │
├────────┼────────┼─────────────────────────────────────────────────────────────┼────────────────────┤
│"4"     │"Sci-Fi"│"Twelve Monkeys (a.k.a. 12 Monkeys) (1995)"                  │"3.8880633977216443"│
└────────┴────────┴─────────────────────────────────────────────────────────────┴────────────────────┘
```

As can be seen, we have successfully created a custom recommendation engine for movies using Python and Neo4j.