Table of Contents

[Work 2](#_Toc118124866)

[Part 1 2](#_Toc118124867)

[Part 2 2](#_Toc118124868)

[Part 3 3](#_Toc118124869)

[Part 4 3](#_Toc118124870)

[Part 5 5](#_Toc118124871)

[Step 6 6](#_Toc118124872)

[Step 7 7](#_Toc118124873)

[Step 8 8](#_Toc118124874)

[Part 9 11](#_Toc118124875)

[A. 11](#_Toc118124876)

[B. 11](#_Toc118124877)

[C. 11](#_Toc118124878)

[D. 12](#_Toc118124879)

[E. 12](#_Toc118124880)

[F. 13](#_Toc118124881)

[References 13](#_Toc118124882)

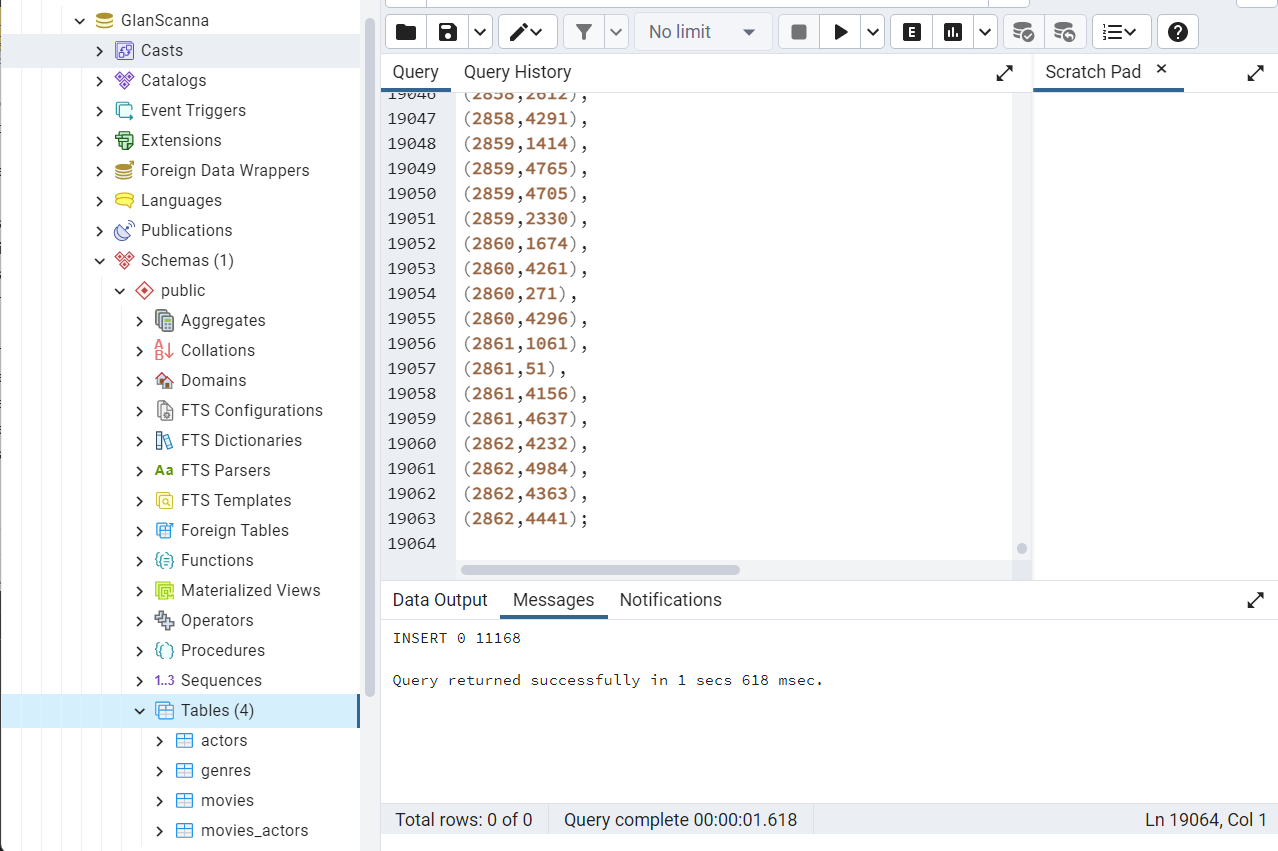
[Appendix 13](#_Toc118124883)

[Appendix 1 – clean\_movies\_actors.py 13](#_Toc118124884)

[Appendix 2 – clean\_movies.py 14](#_Toc118124885)

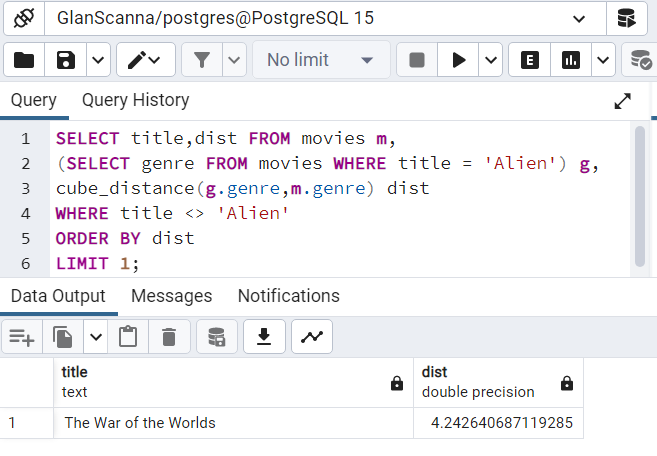
# Work

## Part 1



Here the tables are in the GlanScanna database with the 11168 entries inserted. The cube extension was inserted to add the data into the tables.

## Part 2



This query uses genre to find the most similar movie to Alien. From the bottom up, it limits the results to 1 entry as we only seek the most similar movie. The movies are ordered by “dist”, the distance between the genres. Since the ordering is ascending, the smallest distances will be first. As such showing the most similar movie first. The “where” clause removes the Alien movie from the results as it would always be first, having a distance of 0 to itself.

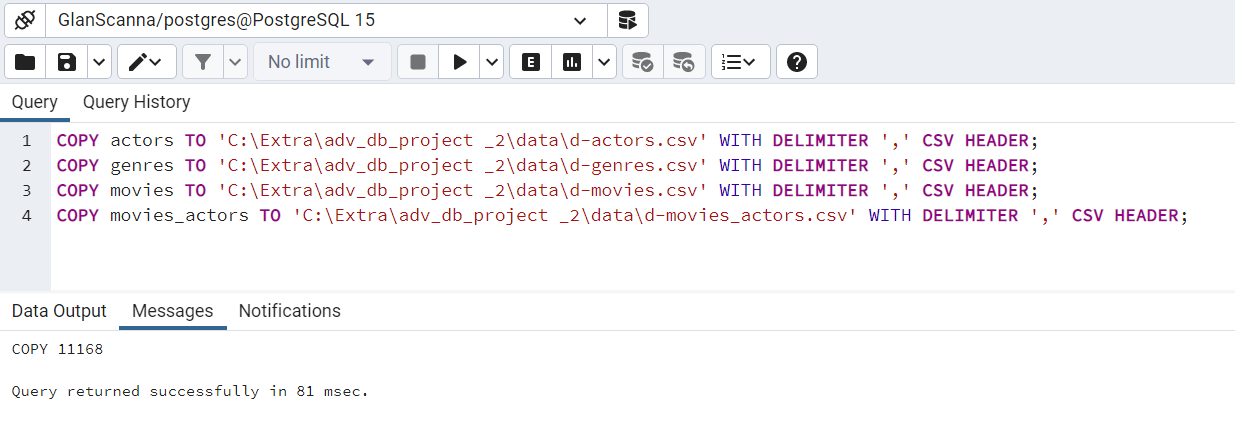
Line 3 creates the distance value, “dist”. It uses the cube\_distance function between the Alien movie’s genre cube and the genre cube of each entry in the movies table. Line 2 creates a reference which contains the genre of the Alien movie. Lastly line 1 sets for the title to be displayed.

## Part 3

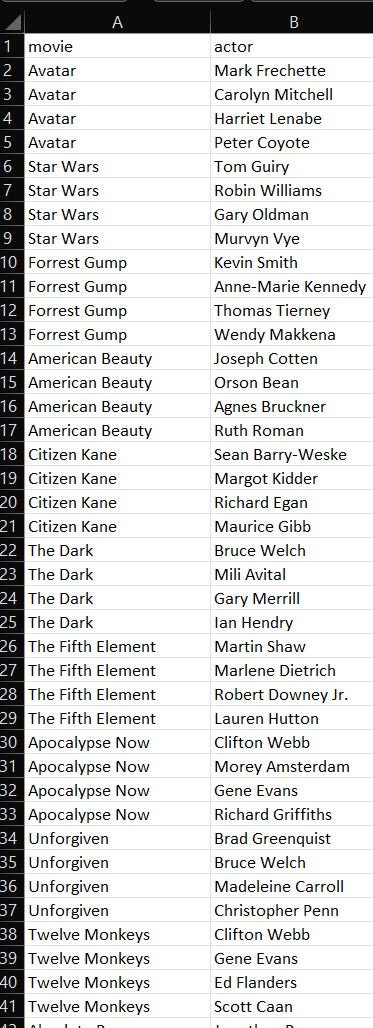
As described by Neo4j (Neo4j, 2022b), data tables will be become the labels in the graph. For the dataset, this includes actors, movies, and genres. The movies\_actors table will be changed to relationships as all the data in this table are connections between actors and movies from the respective tables.

Each row will become a node. For actors, the actor names will be converted to properties on actor nodes. The primary key will be dropped. The genres similarly will each make a genre node with the name of the genre as a property. Movies will become movie nodes. The genre cubes will become many relationships to each of the genre nodes. The weighting of the genre will become a property on the relationship between the movie and genre. Relations with a weighting of 0 will not be created. Last, each row in the movies\_actors will become a relationship.

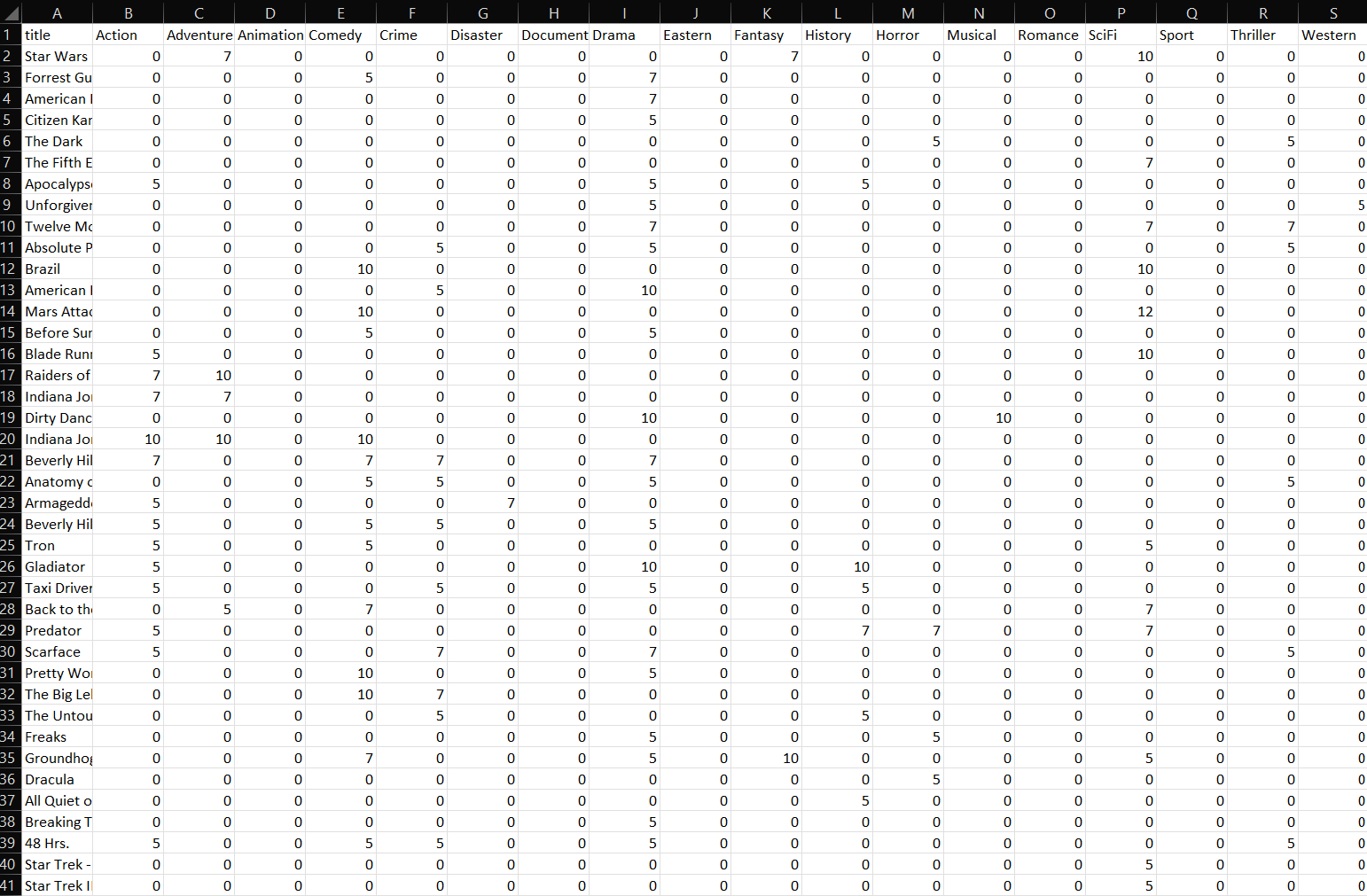
## Part 4



The tables are exported to csv files as described by HEVO (Verma, 2022).



The movies\_actors file relied on indexes which the graph database would not include. The file was run through the clean\_movies\_actors script (Appendix 1) to swap the ID numbers to the titles of the movies and names of the actors respectively.



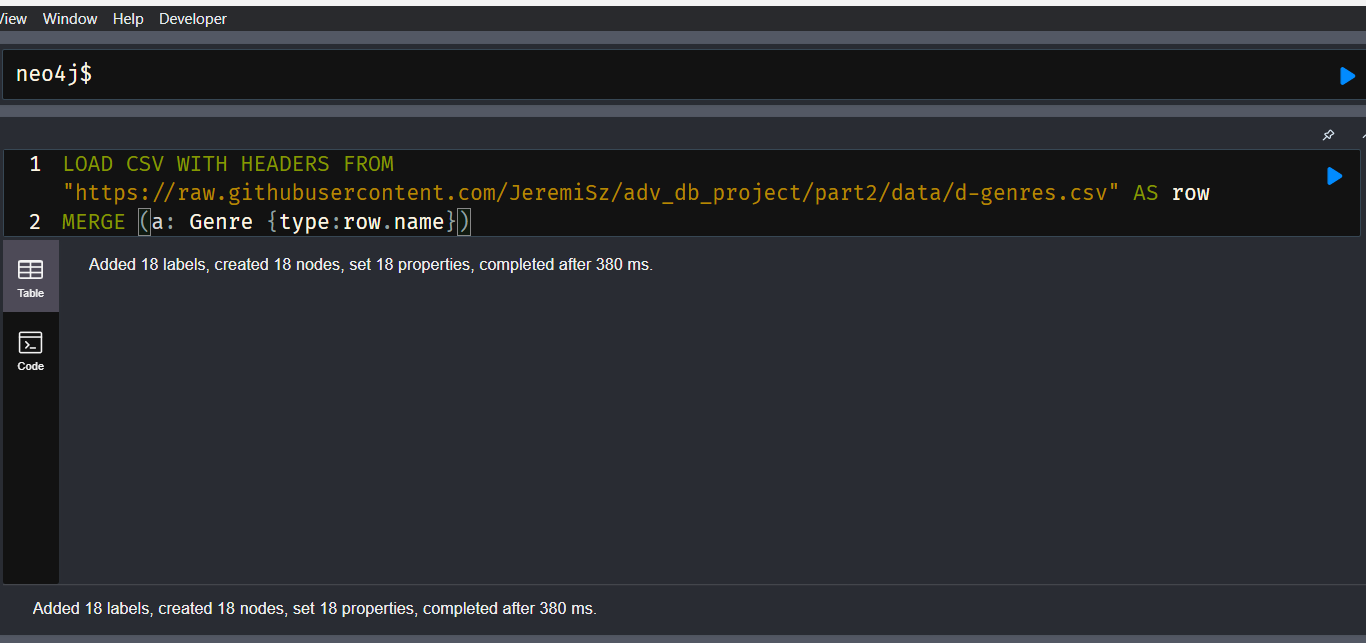
The movies and genres tables are tied via numeric IDs. The clean\_movies script (Appendix 2), removes the numeric keys. It also splits up the cube into individual columns, inserting the genre of each column as the header.

## Part 5

Using Neo4j’s guide on importing CSV files into Neo4j graph database (Neo4j, 2022), actors, movies and genres were imported via the following Cypher commands.



Here the actors file is loaded in from GitHub and Actor nodes are being created with the names of each actor.

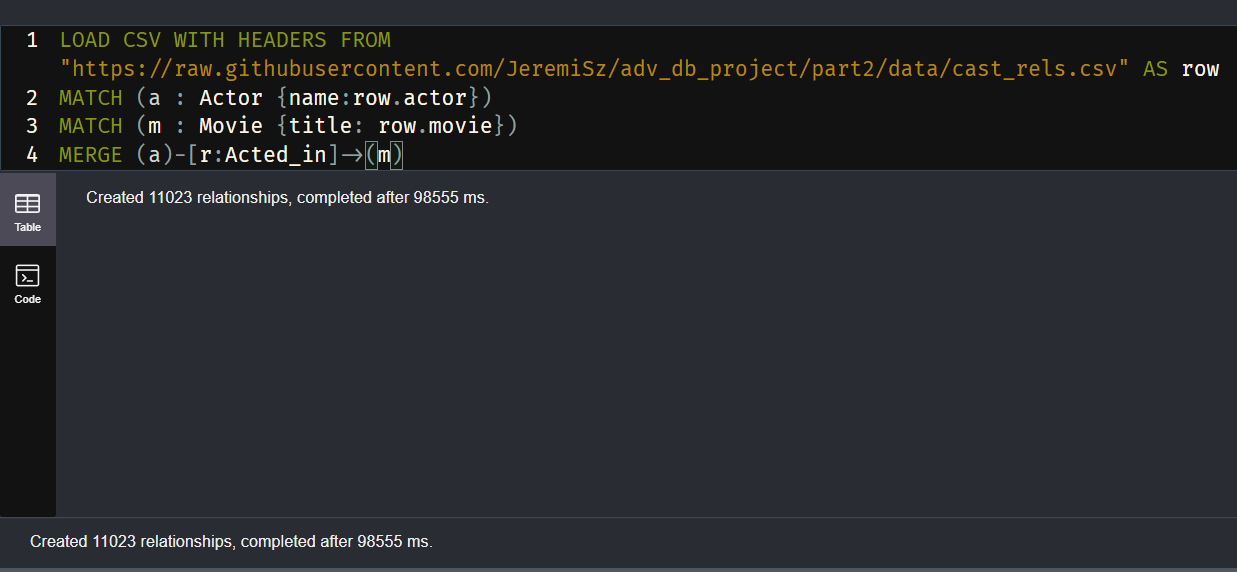


The genres file is loaded, and a genre node is made for each.



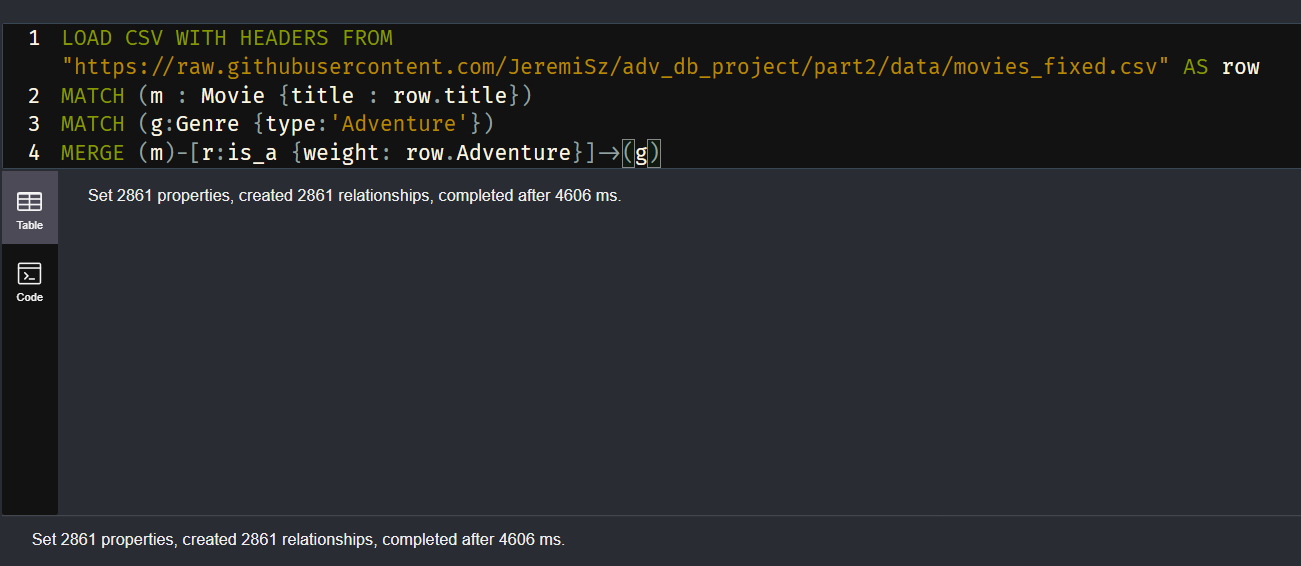
Lastly the same happens to the movies file, creating the movie nodes.

## Step 6

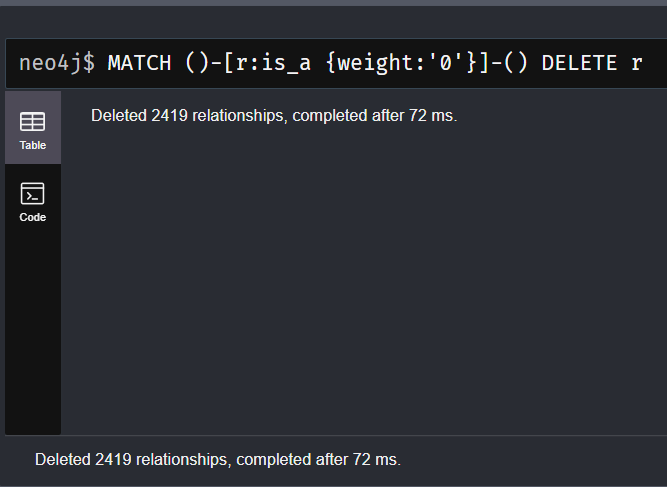


To add the movies actors relationship, the newly created cast\_rels file was loaded. The actor and movie from each row was found with match and a acted\_in relationship was created between them.

## Step 7

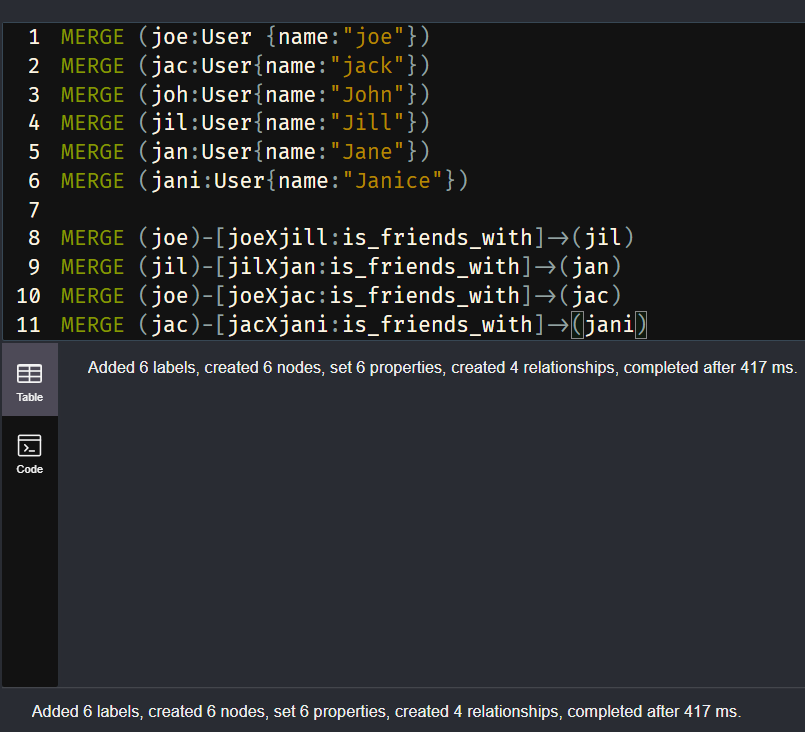


With conditionals being limited to matches and for loops, first every movie genre relationship was created and populated with the weight of the relationship.

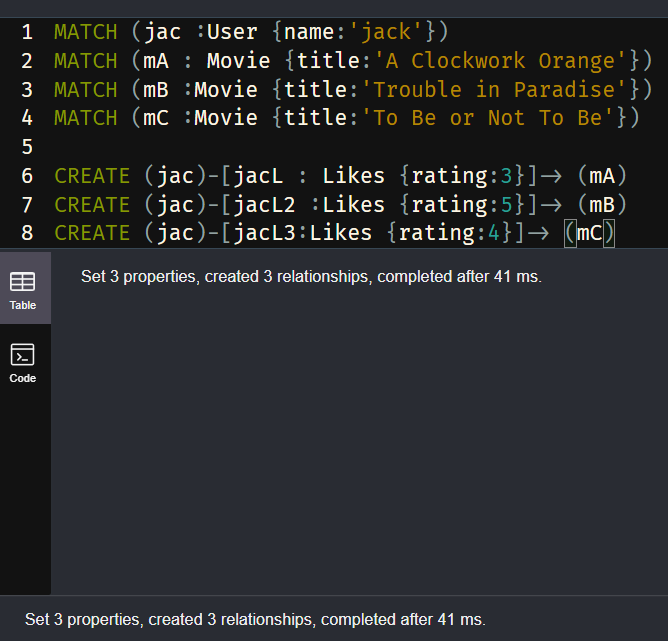


Then the is\_a relationships with a weight of 0 were deleted.

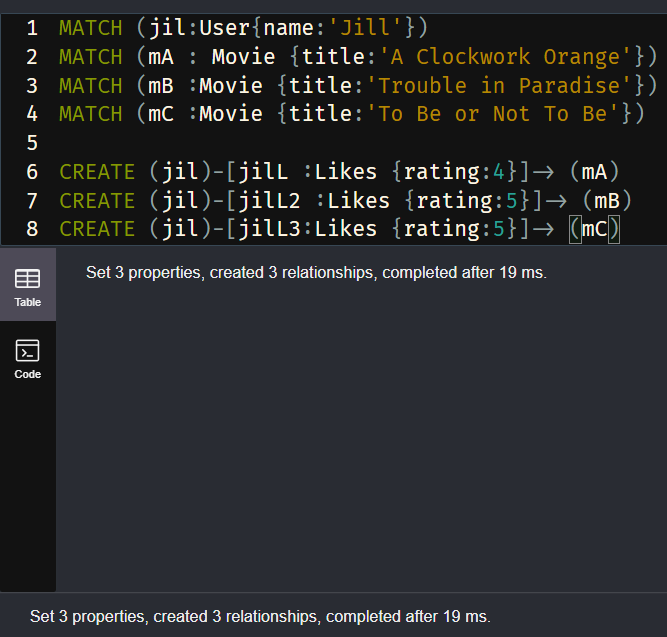
## Step 8



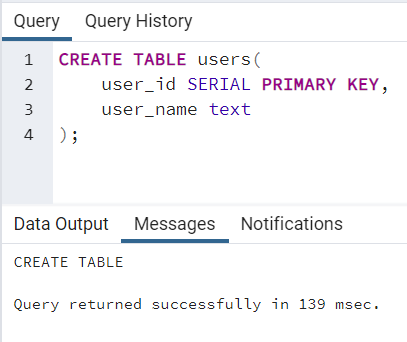
Added 6 users and the friendships between the users.



Matched both Jack the user and 3 movies. Added Jack’s rates for each.



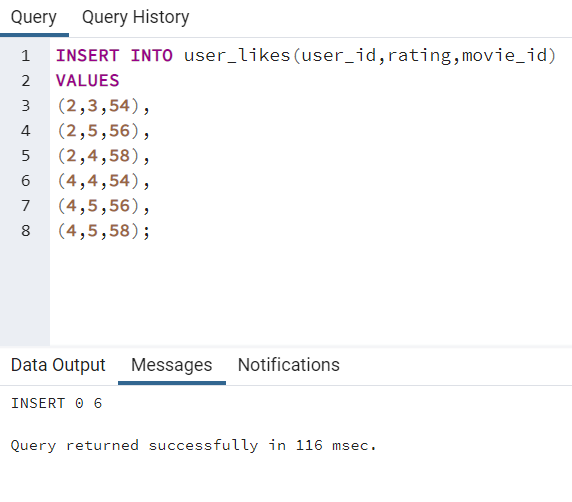
Same for Jill.

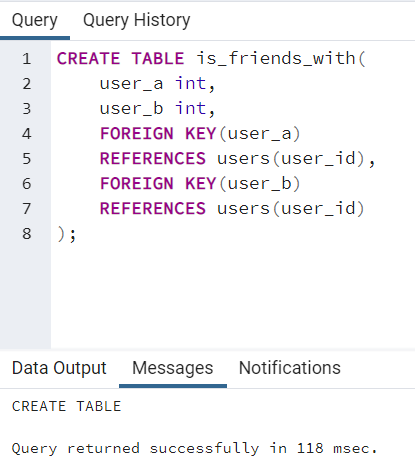


Graphical user interface, text, application, email

Description automatically generated

Text

Description automatically generated

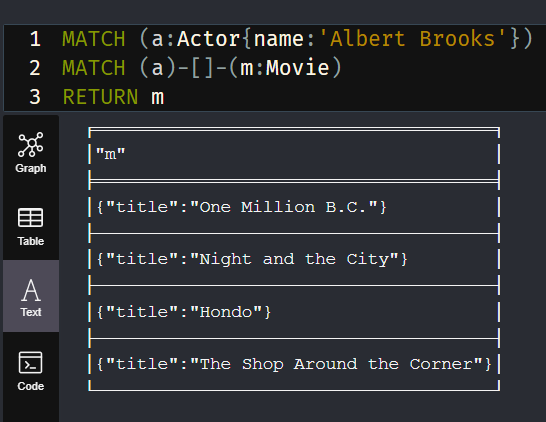


Graphical user interface, text

Description automatically generated

## Part 9

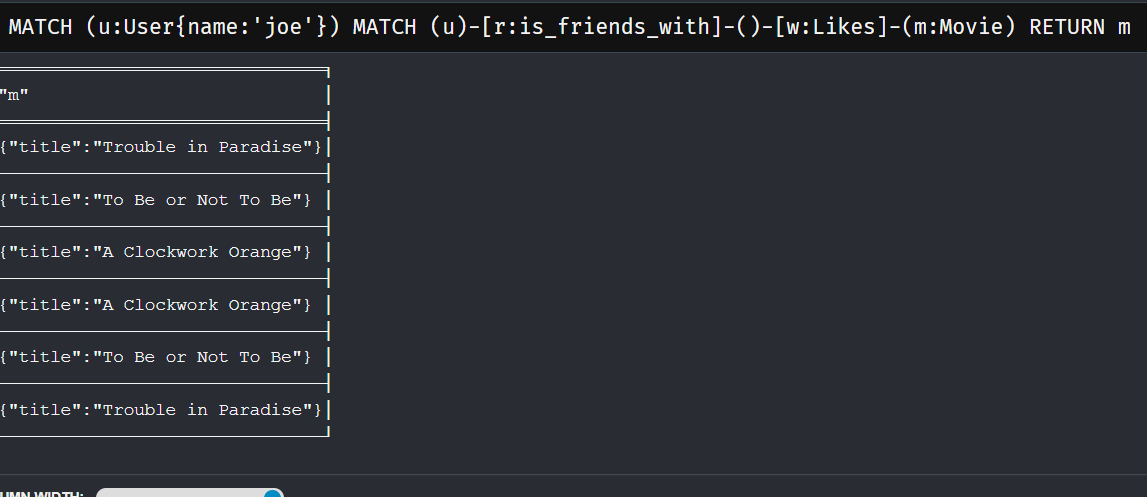
### A.

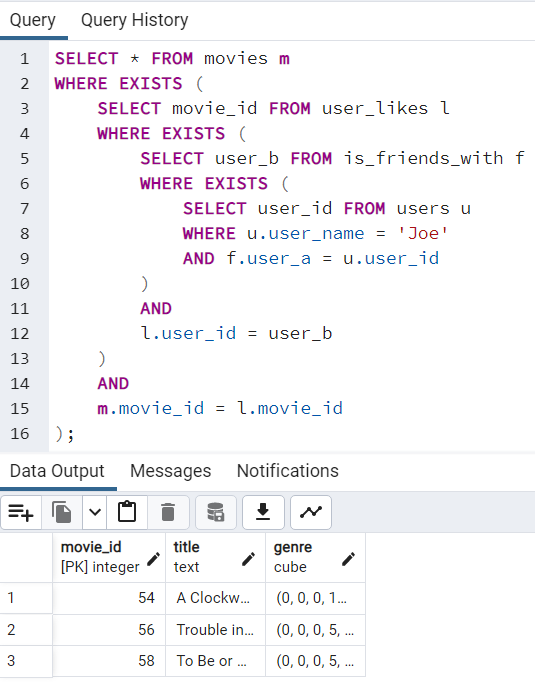


### B.



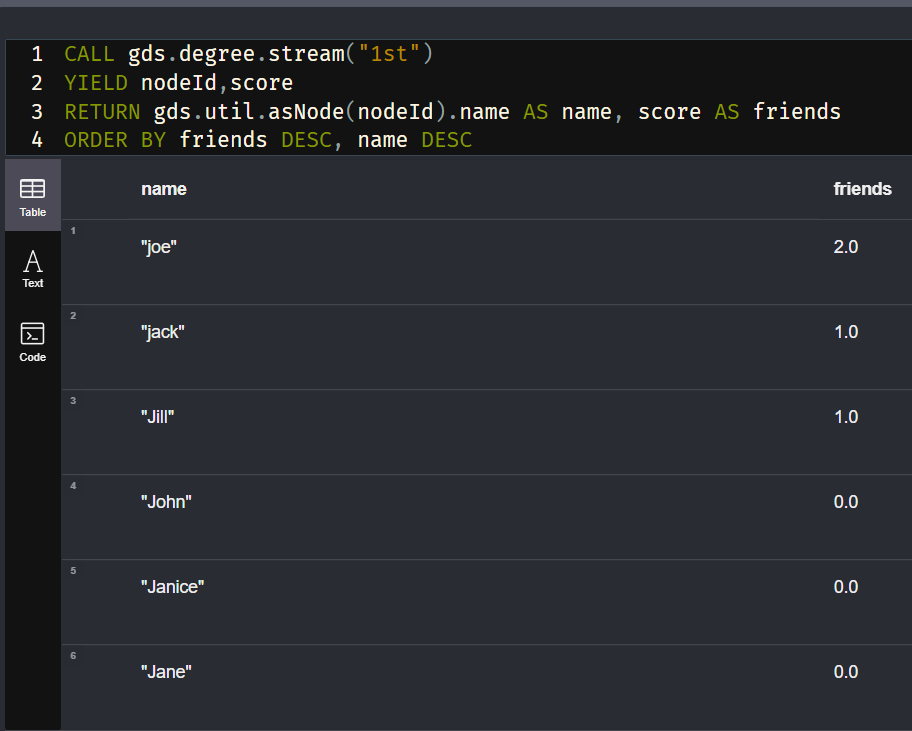
### C.





The relational version is difficult to read and requires unintuitive logic with the EXISTS operator on subqueries. Checks occur a subquery lower than expected. This contrasts with the graph where just the types of the relationship and node types have to specified.

### D.



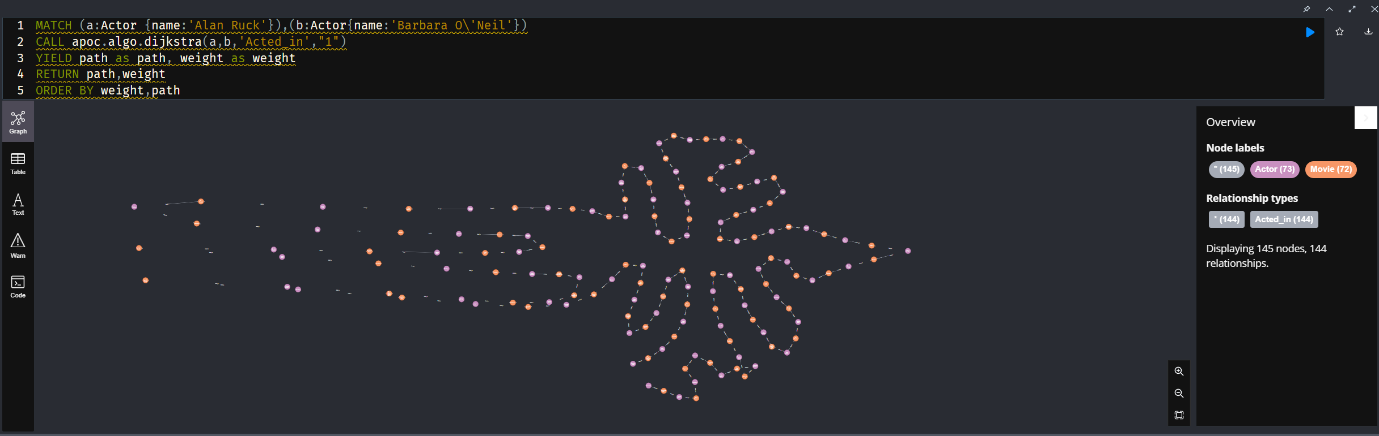
This code first calls on the degree algorithm in the GDS library. As described by the Neo4j site (Neo4j, 2022), this algorithm counts how many connections each node has. Due to prior set up, this only include the is\_friends\_with connection and not the Likes relationship. Then the ID of each node and the score from the algorithm is returned for each node of the type User. Another GDS function translates the node ID to the name property on the nodes. Finally, the results are sorted, 1st by the score, then in alphabetical order.

Text

Description automatically generated

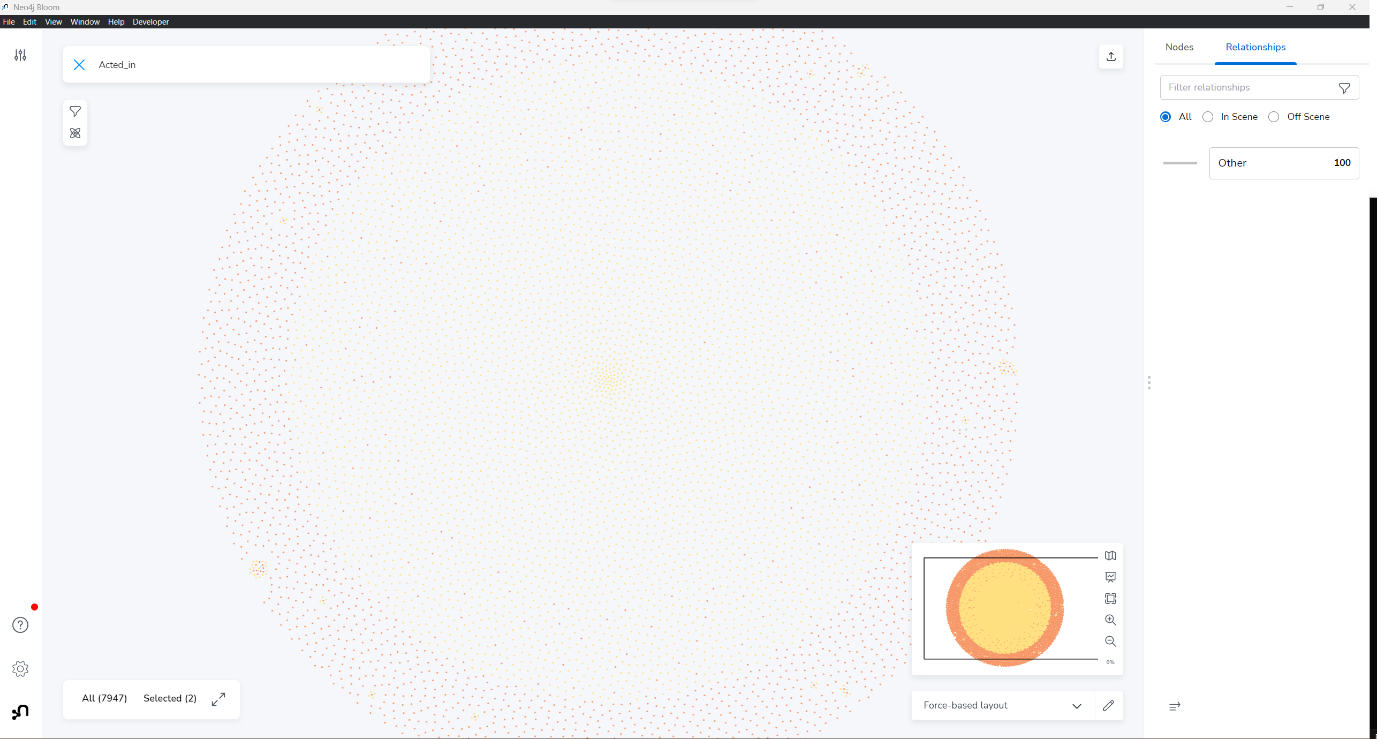
For this implementation, the code describes directly what is happening on the database but not the intent. The code must be read and understood before the whole purpose can be understood. This method also has a join which can be costly on large tables. These joins come purely from the need to use id foreign/primary keys. Using the names directly would increase either look up time or memory space, depending on if the names were linked to or stored with the relationship.

### E.



Done via the method described (Neo4j, 2022a)

### F.



# References

Neo4j, 2022. *Degree Centrality - Neo4j Graph Data Science*. [online] Neo4j Graph Data Platform. Available at: <https://neo4j.com/docs/graph-data-science/2.2/algorithms/degree-centrality/> [Accessed 31 October 2022].

Neo4j, 2022a. *Dijkstra Source-Target Shortest Path - Neo4j Graph Data Science*. [online] Neo4j Graph Data Platform. Available at: <https://neo4j.com/docs/graph-data-science/2.2/algorithms/dijkstra-source-target/> [Accessed 31 October 2022].

Neo4j, 2022. *Importing CSV Data into Neo4j - Developer Guides*. [online] Neo4j Graph Data Platform. Available at: <https://neo4j.com/developer/guide-import-csv/> [Accessed 29 October 2022].

Neo4j, 2022b. *Model: Relational to Graph - Developer Guides*. [online] Neo4j Graph Data Platform. Available at: <https://neo4j.com/developer/relational-to-graph-modeling/> [Accessed 29 October 2022].

Verma, R., 2022. Postgres Export to CSV: Best Ways & Steps to Export Data - Learn | Hevo. Available at: <https://hevodata.com/learn/postgres-export-to-csv/> [Accessed 29 October 2022].

# Appendix

## Appendix 1 – clean\_movies\_actors.py

import csv

DATA\_LOC = "../data/"

SEP = ","

rels\_file = open(DATA\_LOC + "d-movies\_actors.csv")

movies\_file = open(DATA\_LOC + "d-movies.csv")

actors\_file = open(DATA\_LOC + "d-actors.csv")

rels = rels\_file.readlines()[1:]

movies = movies\_file.readlines()[1:]

actors = actors\_file.readlines()[1:]

rels\_file.close()

movies\_file.close()

actors\_file.close()

data = []

for rel in rels:

    movie\_id, actor\_id = rel.split(sep=SEP)

    movie\_id = int(movie\_id) - 2

    actor\_id = int(actor\_id) - 2

    movie = movies[movie\_id].split(sep=SEP)[1]

    actor = actors[actor\_id].split(sep=SEP)[1]

    data.append([movie,actor[:len(actor)-1]])

fixed\_rels = open(DATA\_LOC + "cast\_rels.csv","w",newline='', encoding='utf-8')

rels\_file = csv.writer(fixed\_rels)

rels\_file.writerow(["movie","actor"])

rels\_file.writerows(data)

fixed\_rels.close()

This code assumes the data will be in a data folder in the parent folder of the current folder. The movies\_actors data is assumed to be in the file “d-movies\_actors.csv”, the movies in “d-movies.csv” and actors in “d-actors.csv”

## Appendix 2 – clean\_movies.py

import csv

DATA\_LOC = "../data/"

SEP = ","

movies\_file = open(DATA\_LOC + "d-movies.csv")

genres\_file = open(DATA\_LOC + "d-genres.csv")

movies = csv.reader(movies\_file)

genres = genres\_file.readlines()[1:]

genres\_file.close()

#make template

genre\_list = []

for genre in genres:

    genre = genre.split(sep=SEP)[0]

    genre\_list.append(genre)

movie\_data = []

headers = []

for movie in movies:

    title = movie[1]

    if title == "title":

        headers.append(title)

    else:

        genre\_weight = movie[2]

        genre\_weight = genre\_weight[1:len(genre\_weight)-1]

        genre\_weight = genre\_weight.split(sep=SEP)

        data = [title] + genre\_weight

        movie\_data.append(data)

movies\_file.close()

headers = headers + genre\_list

movies\_file = open(DATA\_LOC + "movies\_fixed.csv","w",newline='', encoding='utf-8')

movies = csv.writer(movies\_file)

movies.writerow(headers)

movies.writerows(movie\_data)

movies\_file.close()

This code assumes the data will be in a data folder in the parent folder of the current folder. genre data is assumed to be in the file “d-genres.csv”, and the movies in “d-movies.csv”.