# Creating the database in PostgreSQL

create database project2;



Figure 1 Create database in Postgres

# 2. Using the cube,

SELECT title, cube\_distance(genre, '(0,0,0,0,0,0,0,0,0,0,0,10,0,0,10,0,0,0)') dist FROM movies WHERE cube\_enlarge('(0,0,0,0,0,0,0,0,0,0,0,10,0,0,10,0,0,0)'::cube, 5, 18) @> genre ORDER BY dist;



Figure 2.1 Using Cube

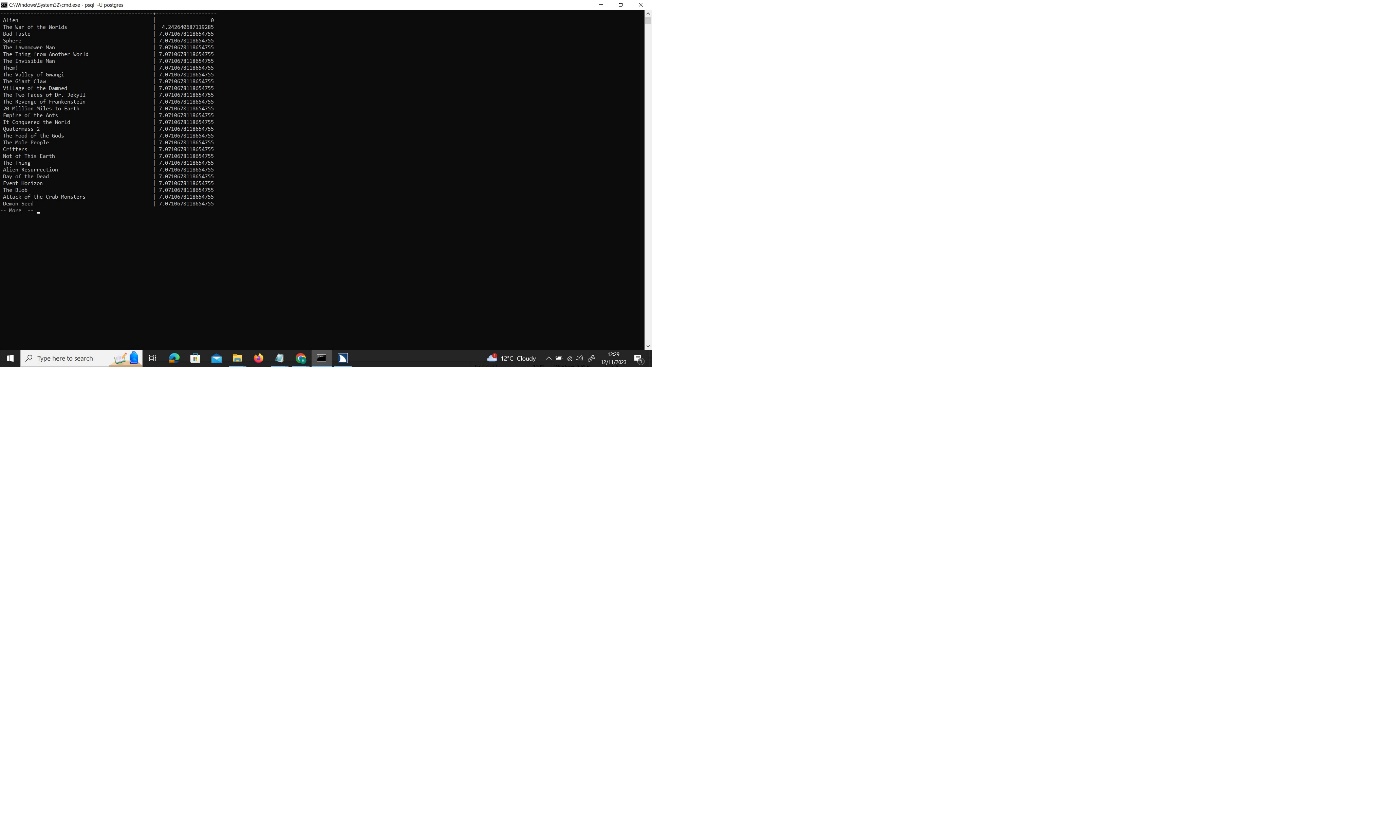


Figure 2.2 Alien movies

# 3. Strategy for converting from relational model to graph model

On the Graph databases data and relationships are equally important. These databases build sophisticated and simple models that are close to the business domain. The data (node) does not suffer too much change from the relational databases, in order to be able to query and view the data on an understandable way. Nodes contain a list of relationships to other nodes, and these relationships may have several types, attributes and directions:

Directed: one direction.

Undirected: both directions (Neo4J, 2023) .

Graph databases use the relationships to direct access the connected nodes. This is more efficient in the case of Join-heavy queries, since the query is faster on a graph database.

The process to convert a relational database to a graph database follow the following procedure (Neo4J, 2023) -

1- The table is converted into a node label

2- Each row is converted into a node.

3. Each column is converted into a node attribute.

4- Technical primary keys are deleted. However, business keys are preserved.

5- Adding indexes for frequent lookup attributes i.e., business primary keys.

6- Foreign keys are converted into relationships.

7- No need for default values on the attributes.

8- Duplicated data must be stored in separated nodes.

9- Index columns are converted into arrays.

10- Join tables are converted into relationships.

# 4. Exporting data into CSV

The modification made was on the Movies table. The cube field was extracted and moved to a new csv file called genres\_movies.csv. On this new file each value is turned into an individual column. In addition, each row contains the corresponding movie ID.

\COPY genres TO 'C:\Users\TEST\Documents\ITT\Forth\_Year\AdvancedDatabases\Project2\genres.csv' DELIMITER ',' CSV HEADER ENCODING 'UTF8';

\COPY movies TO 'C:\Users\TEST\Documents\ITT\Forth\_Year\AdvancedDatabases\Project2\movies.csv' DELIMITER ',' CSV HEADER ENCODING 'UTF8';

\COPY actors TO 'C:\Users\TEST\Documents\ITT\Forth\_Year\AdvancedDatabases\Project2\actors.csv' DELIMITER ',' CSV HEADER ENCODING 'UTF8';

\COPY movies\_actors TO 'C:\Users\TEST\Documents\ITT\Forth\_Year\AdvancedDatabases\Project2\movies\_actors.csv' DELIMITER ',' CSV HEADER ENCODING 'UTF8';

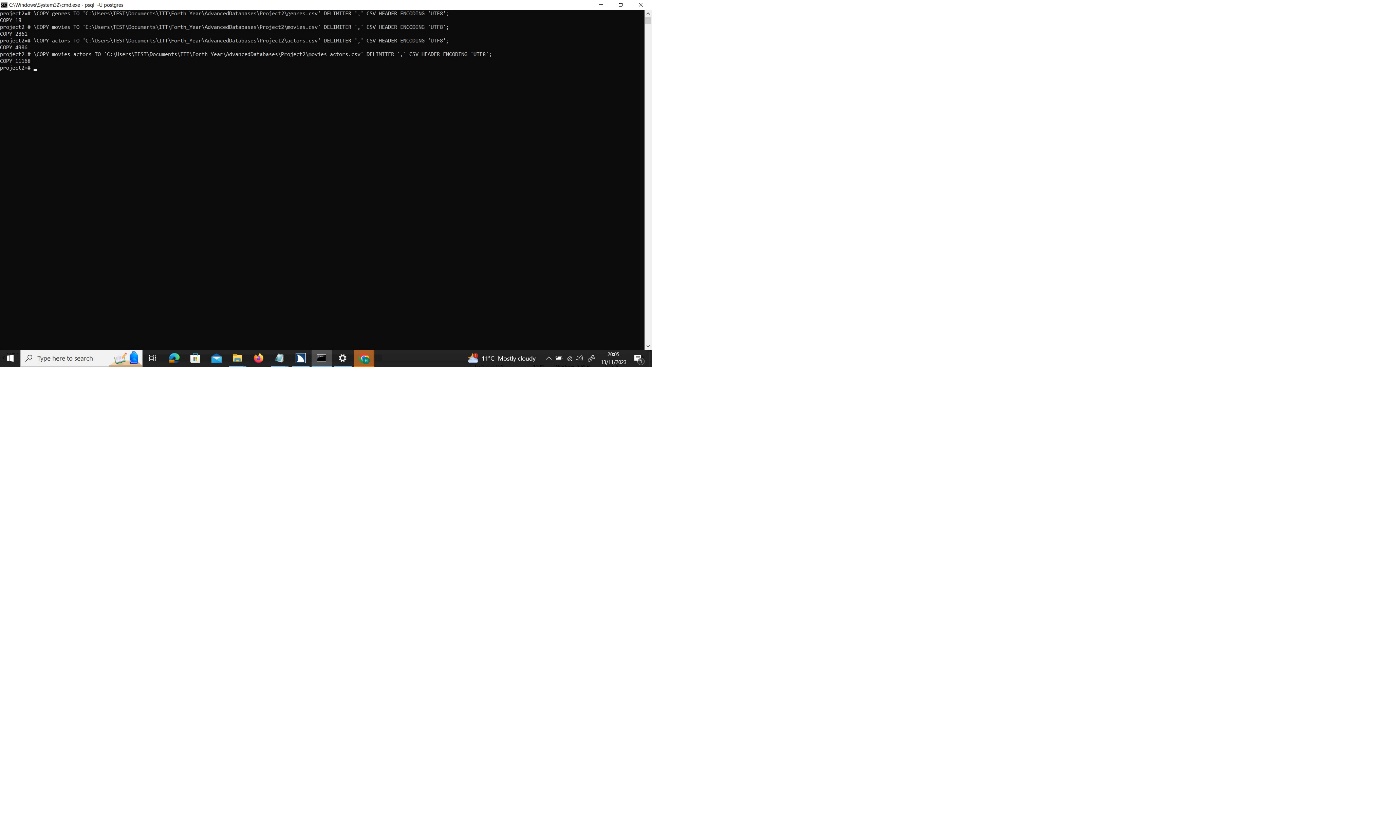


Figure 4 genres\_movies csv file

# 5. Importing movies, actors and genres

LOAD CSV WITH HEADERS from 'file:///genres.csv' AS row

MERGE ( g:Genre { id : row . id })

SET g . name = row . name ;

LOAD CSV WITH HEADERS FROM 'file:///actors.csv' AS row

MERGE (a:Actor {id:row.actor\_id})

SET a.name = row.name;

LOAD CSV WITH HEADERS from 'file:///movies.csv' AS row

MERGE ( m:Movie { id : row . movie\_id })

SET m . title = row . title ;

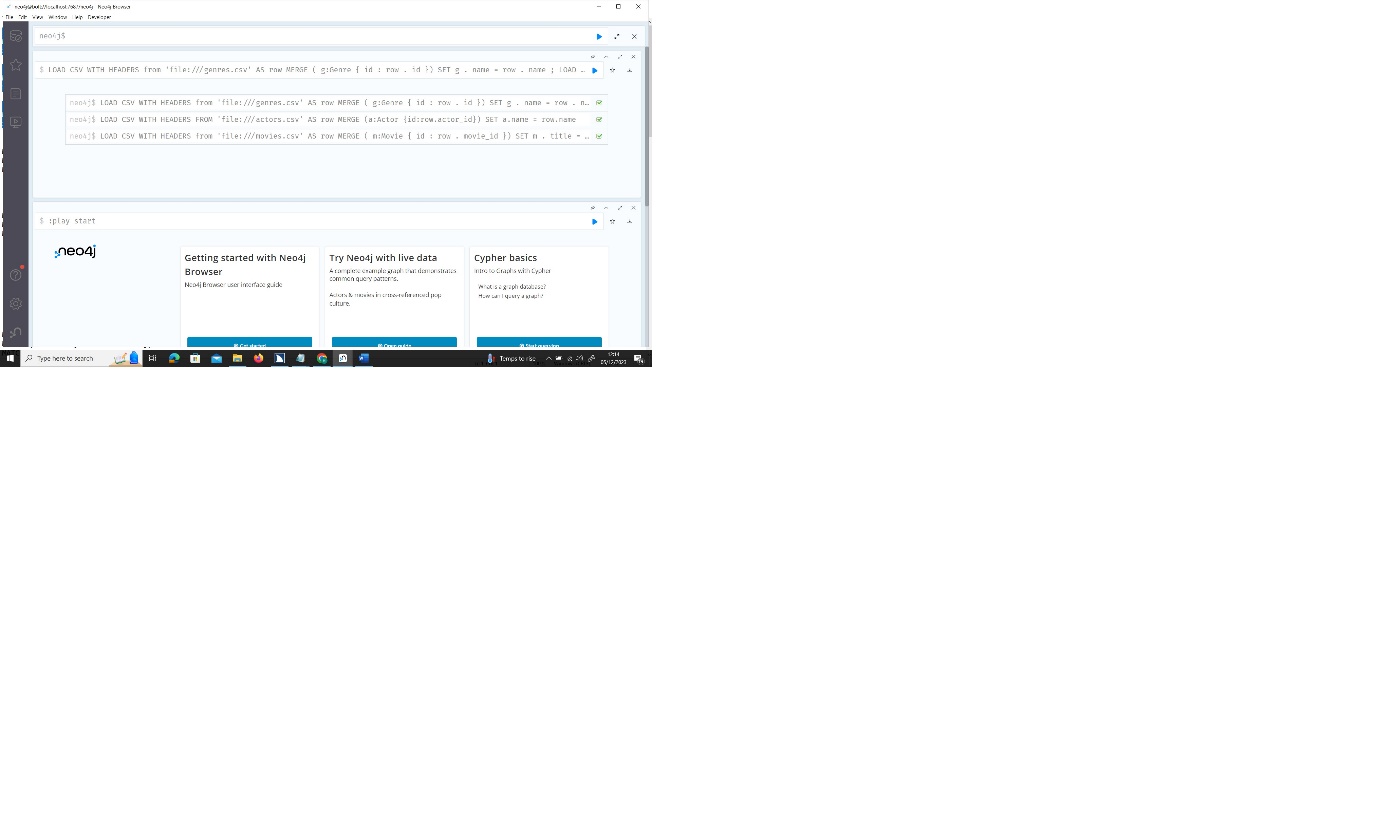


Figure 5 Importing Actors, movies and genres

# 6. Creating relationships between actors and movies

LOAD CSV WITH HEADERS from 'file:///movies\_actors.csv' AS row

MATCH (m:Movie {id: row.movie\_id})

MATCH (a:Actor {id: row.actor\_id})

MERGE (m)-[:PERFORMS]->(a);

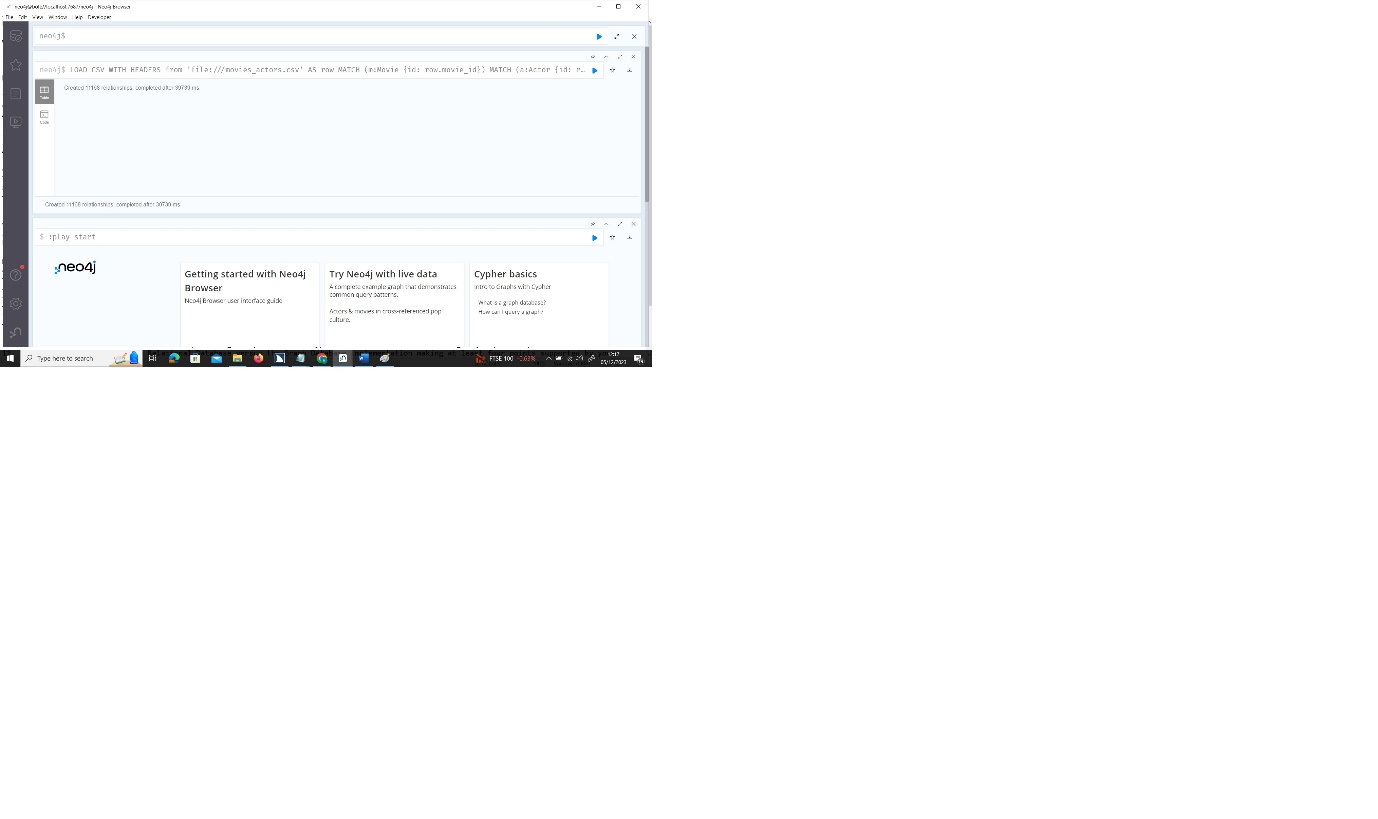


Figure 6 Importing actor and movierelationships

# 7. Creating relationships between movies and genres

The file created on the exercise 4 called genres\_movies.csv is used for this exercise. First the code matches each movie ID along with all the values that correspond with each genre. Then, one relationship is created that links the movie to the genre with the corresponding value.

LOAD CSV WITH HEADERS from 'file:///genres\_movies.csv' AS row

MATCH (m:Movie {id: row.movie\_id})

MATCH (ac:Genre {name: 'Action'})

MATCH (ad:Genre {name: 'Adventure'})

MATCH (an:Genre {name: 'Animation'})

MATCH (co:Genre {name: 'Comedy'})

MATCH (cr:Genre {name: 'Crime'})

MATCH (di:Genre {name: 'Disaster'})

MATCH (do:Genre {name: 'Documentary'})

MATCH (dr:Genre {name: 'Drama'})

MATCH (ea:Genre {name: 'Eastern'})

MATCH (fa:Genre {name: 'Fantasy'})

MATCH (hi:Genre {name: 'History'})

MATCH (ho:Genre {name: 'Horror'})

MATCH (mu:Genre {name: 'Musical'})

MATCH (ro:Genre {name: 'Romance'})

MATCH (sc:Genre {name: 'SciFi'})

MATCH (sp:Genre {name: 'Sport'})

MATCH (th:Genre {name: 'Thriller'})

MATCH (we:Genre {name: 'Western'})

MERGE (m)-[:HAS {weight : toInteger(row.Action)}]->(ac)

MERGE (m)-[:HAS {weight : toInteger(row.Adventure)}]->(ad)

MERGE (m)-[:HAS {weight : toInteger(row.Animation)}]->(an)

MERGE (m)-[:HAS {weight : toInteger(row.Comedy)}]->(co)

MERGE (m)-[:HAS {weight : toInteger(row.Crime)}]->(cr)

MERGE (m)-[:HAS {weight : toInteger(row.Disaster)}]->(di)

MERGE (m)-[:HAS {weight : toInteger(row.Documentary)}]->(do)

MERGE (m)-[:HAS {weight : toInteger(row.Drama)}]->(dr)

MERGE (m)-[:HAS {weight : toInteger(row.Eastern)}]->(ea)

MERGE (m)-[:HAS {weight : toInteger(row.Fantasy)}]->(fa)

MERGE (m)-[:HAS {weight : toInteger(row.History)}]->(hi)

MERGE (m)-[:HAS {weight : toInteger(row.Horror)}]->(ho)

MERGE (m)-[:HAS {weight : toInteger(row.Musical)}]->(mu)

MERGE (m)-[:HAS {weight : toInteger(row.Romance)}]->(ro)

MERGE (m)-[:HAS {weight : toInteger(row.SciFi)}]->(sc)

MERGE (m)-[:HAS {weight : toInteger(row.Sport)}]->(sp)

MERGE (m)-[:HAS {weight : toInteger(row.Thriller)}]->(th)

MERGE (m)-[:HAS {weight : toInteger(row.Western)}]->(we);

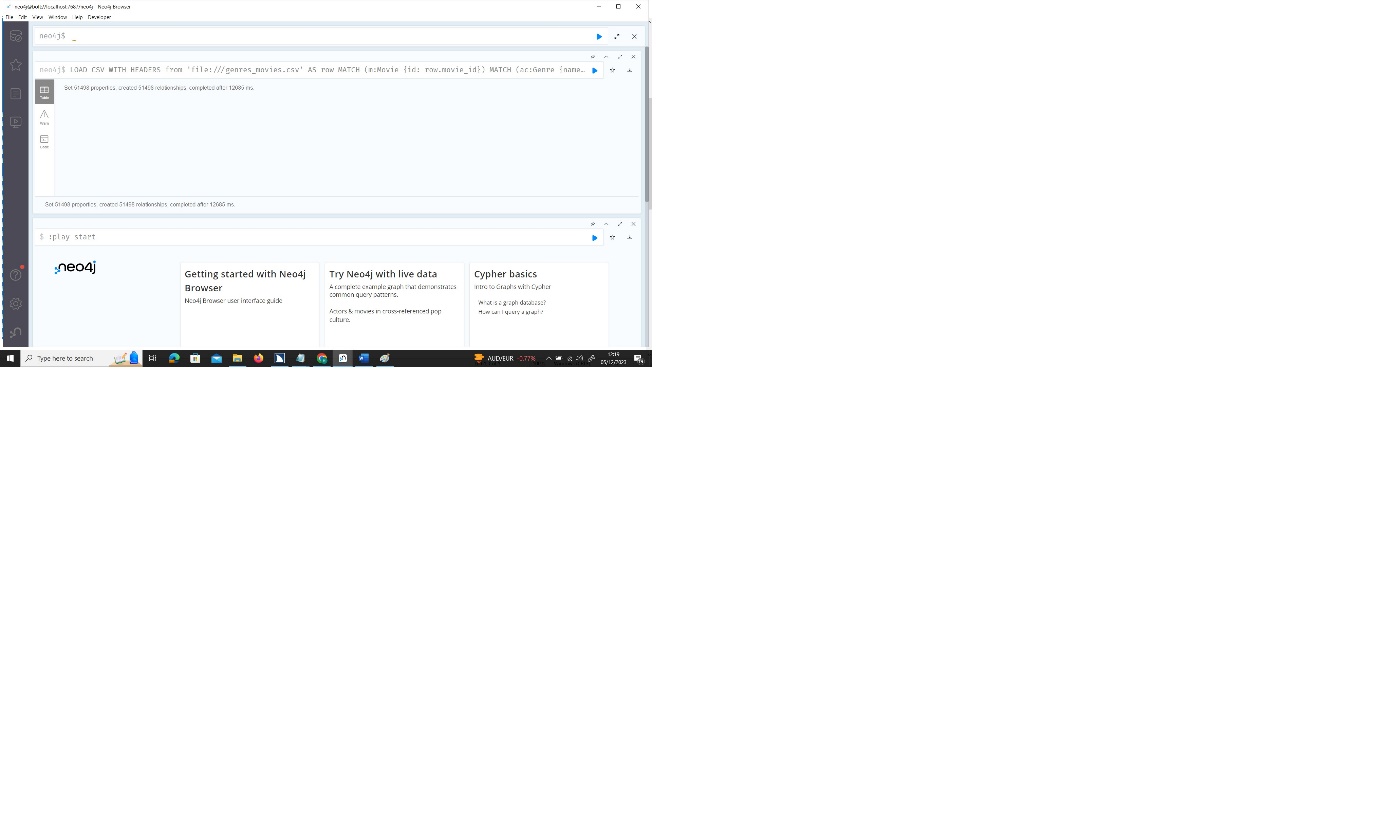


Figure 7 Importing genre movie relationships

# 8. Adding the social media section

## 8.1 Graph code

CREATE (joe:person{name:'Joe'}),

(jack:person{name:'Jack'}),

(john:person{name:'John'}),

(jill:person{name:'Jill'}),

(jane:person{name:'Jane'}),

(janice:person{name:'Janice'});

MATCH (joe:person{name:'Joe'}),

(jill:person{name:'Jill'})

CREATE (joe)-[r:isfriend]->(jill);

MATCH (jill:person{name:'Jill'}),

(jane:person{name:'Jane'})

CREATE (jill)-[r:isfriend]->(jane);

MATCH (joe:person{name:'Joe'}),

(jack:person{name:'Jack'})

CREATE (joe)-[r:isfriend]->(jack);

MATCH (jack:person{name:'Jack'}),

(janice:person{name:'Janice'})

CREATE (jack)-[r:isfriend]->(janice);

MATCH (jack:person{name:'Jack'}),

(m:Movie{id:'1753'})

CREATE (jack)-[r:LIKES {rate: toInteger('3')}]->(m);

MATCH (jack:person{name:'Jack'}),

(m:Movie{id:'1080'})

CREATE (jack)-[r:LIKES {rate: toInteger('5')}]->(m);

MATCH (jack:person{name:'Jack'}),

(m:Movie{id:'1259'})

CREATE (jack)-[r:LIKES {rate: toInteger('4')}]->(m);

MATCH (jill:person{name:'Jill'}),

(m:Movie{id:'19'})

CREATE (jill)-[r:LIKES {rate: toInteger('5')}]->(m);

MATCH (jill:person{name:'Jill'}),

(m:Movie{id:'284'})

CREATE (jill)-[r:LIKES {rate: toInteger('4')}]->(m);

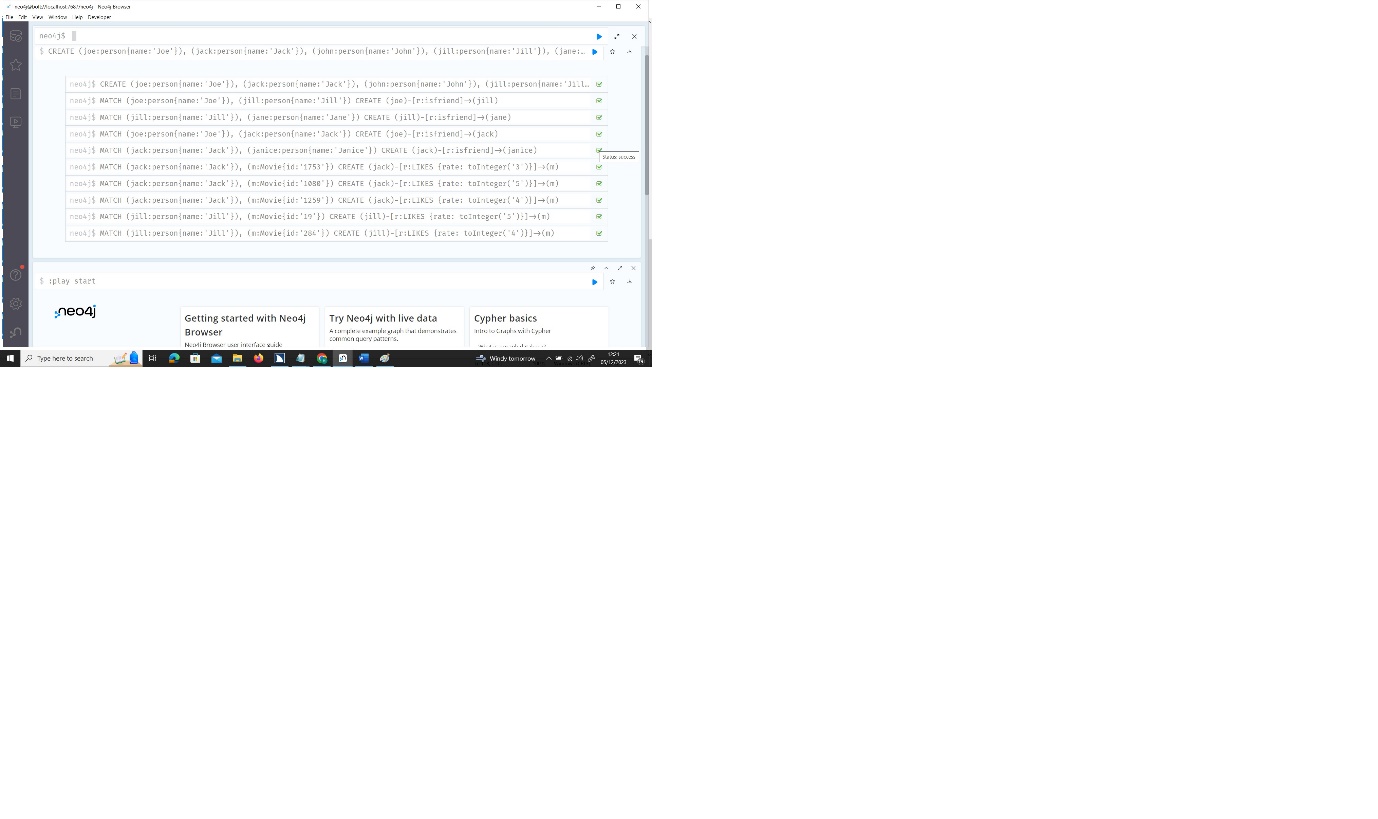


Figure 8 Creating social media

## 8.2 Relational code

CREATE TABLE People (Id SERIAL PRIMARY KEY, Name text);

CREATE TABLE Friends (Person1 integer, person2 integer, FOREIGN KEY (Person1) REFERENCES People(Id), FOREIGN KEY (Person2) REFERENCES People(Id) );

CREATE TABLE Likes (PersonId integer, MovieId integer, Rate integer, FOREIGN KEY (PersonId) REFERENCES People(Id), FOREIGN KEY (MovieId) REFERENCES Movies(movie\_id) );

INSERT INTO People (Name) VALUES

('Joe'), ('Jack'), ('John'), ('Jill'), ('Jane'), ('Janice');

INSERT INTO Friends VALUES

(1, 4), (4, 5), (1, 2), (2, 6);

INSERT INTO Likes VALUES

(2, 1753, 3), (2, 1080, 5), (2, 1259, 4), (4, 1259, 4), (4, 19, 5), (4, 284, 5);

# 9. Querying Neo4J

## 9.1 Movies of an actor

The selected actor is Bruce Willis.

MATCH (a:Actor {name: "Bruce Willis"})--(m:Movie)

RETURN (m.title);

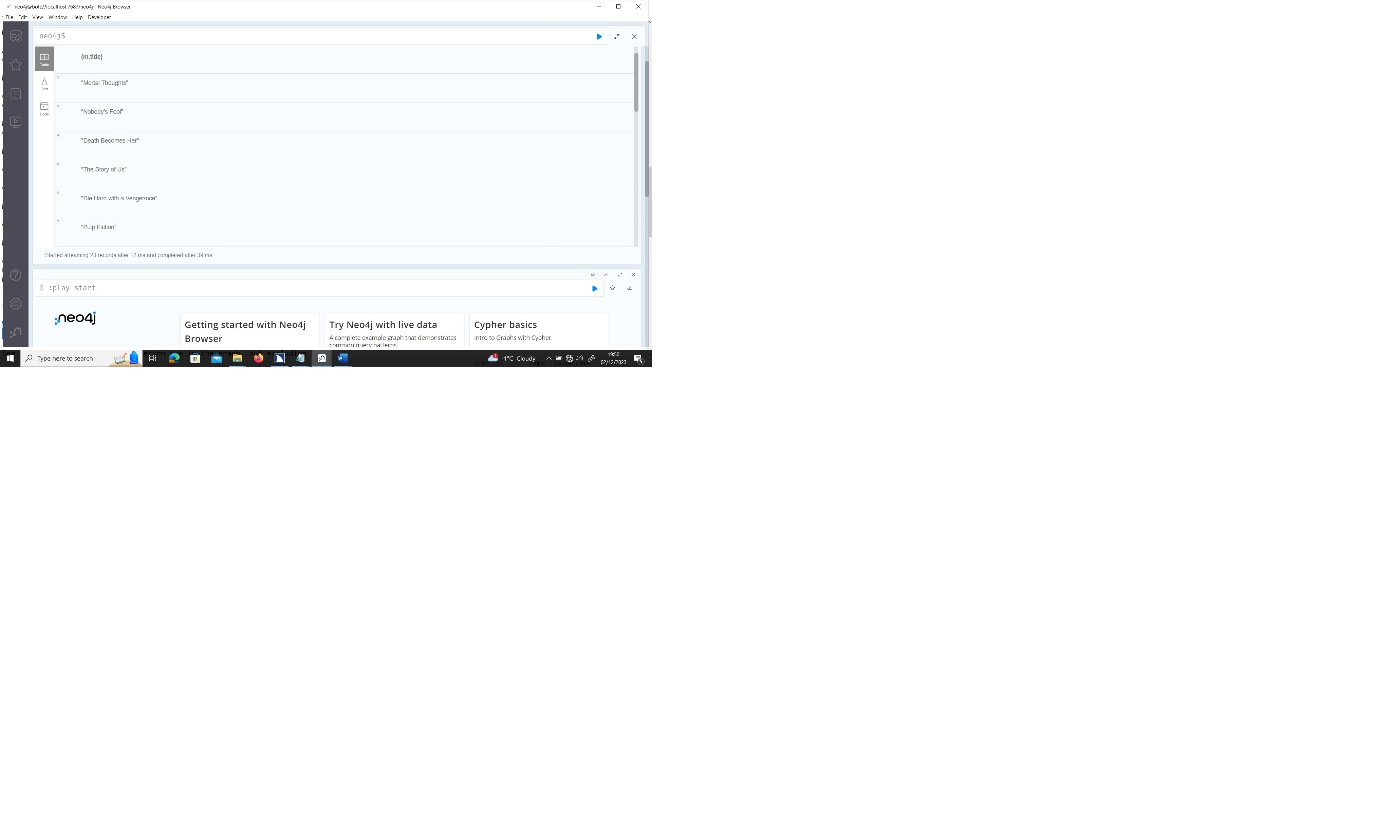


Figure 9.1 movies of an actor

## 9.2 Genres of Movies of the actor

MATCH (g:Genre)-[:HAS]-(m:Movie)-[:PERFORMS]-(a:Actor {name: "Bruce Willis"})

RETURN (g.name);

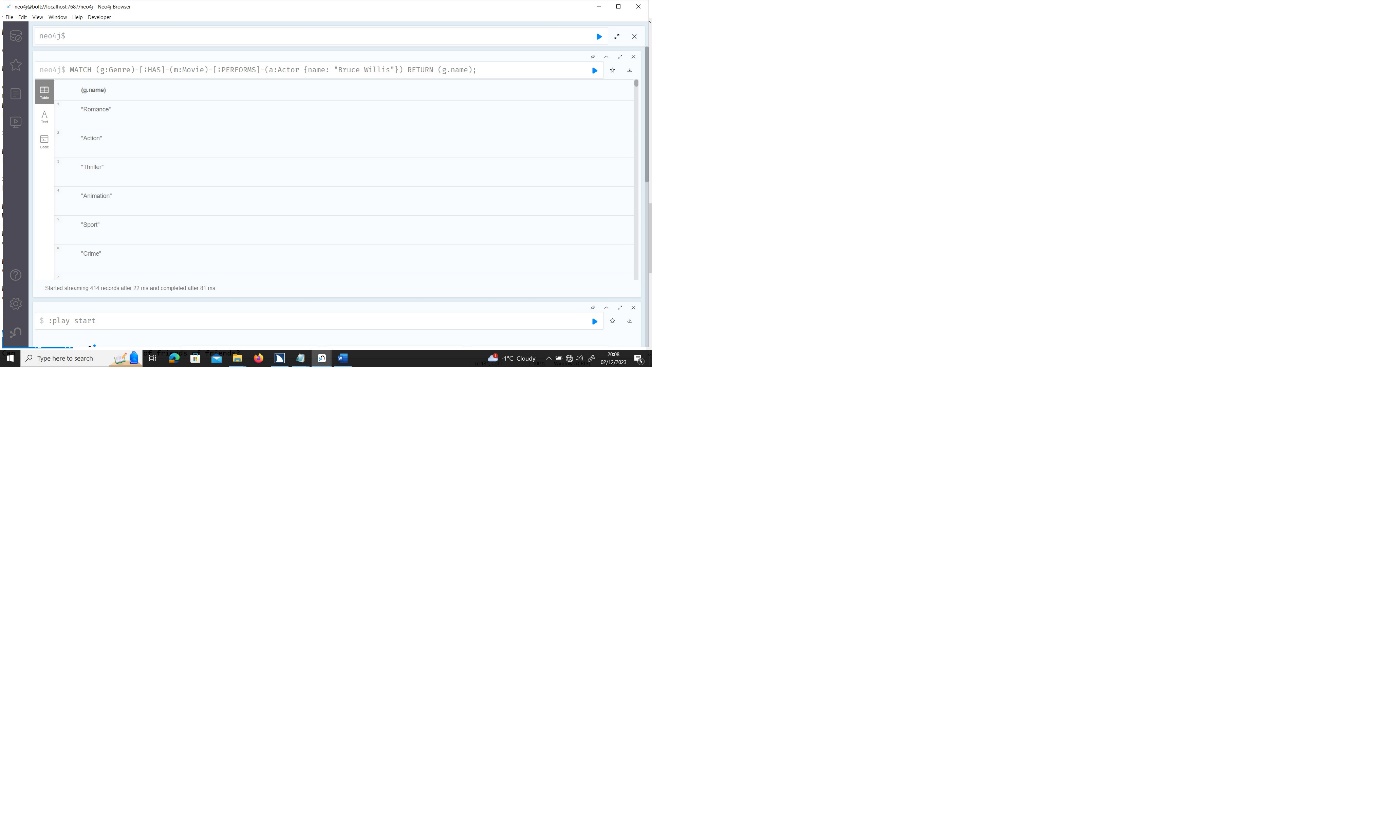


Figure 9.2 Genres of movies of an actor

## 9.3 Movies that Joe’s friends like

MATCH (m:Movie)-[:LIKES]-(p:person)-[:isfriend]-(p1:person {name: "Joe"})

RETURN (m.title);

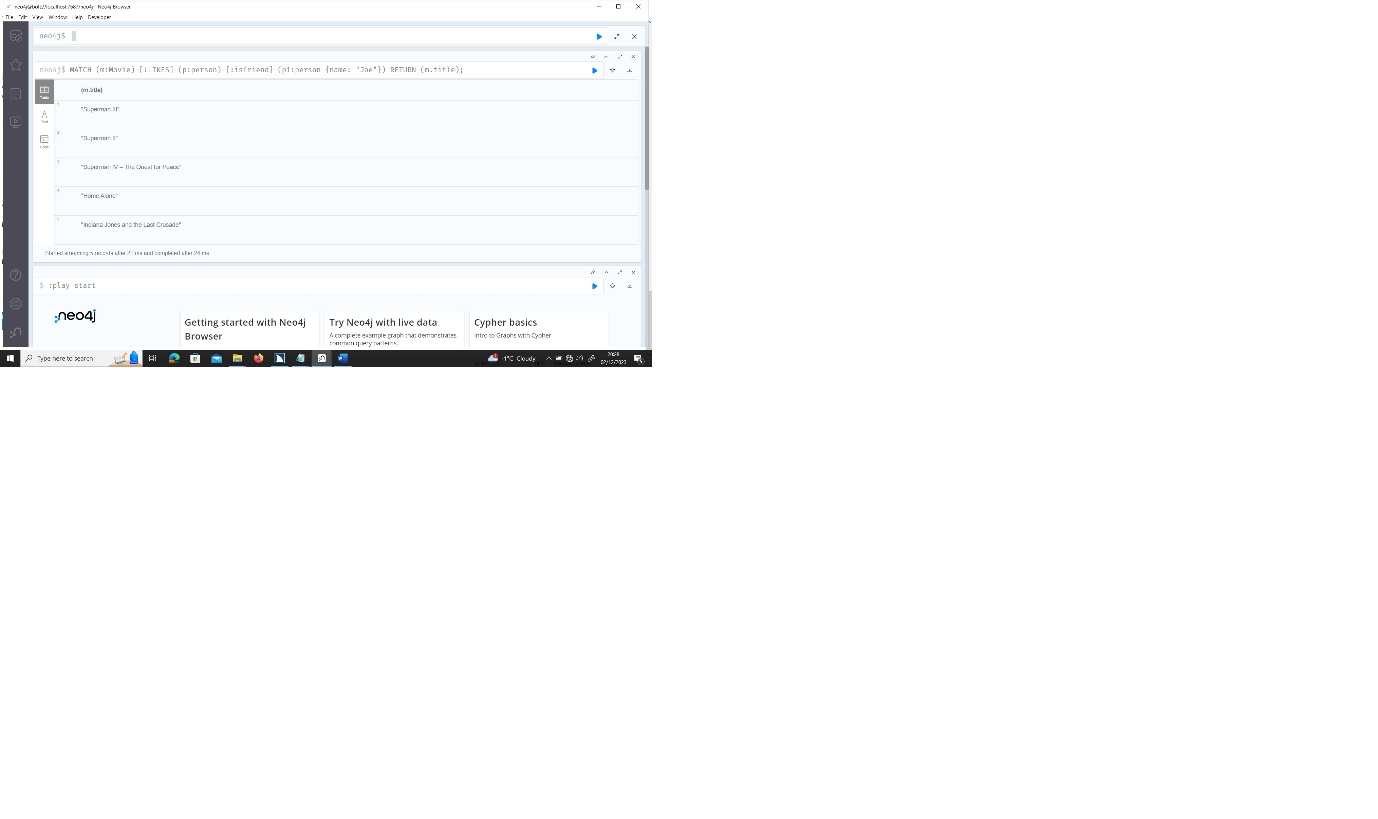


Figure 9.3 Joe friends’ movies

In a relational database, this operation would be done by using the Join statement, along with the where statement.

On the from statement four tables would be used: People, Friends, Likes and Movies. The People table would be joined to the Friends table through the person ID on the People table and Person1 on Friends table and with the OR command to Person2, since the required person ID can be in both fields. Then the Friends table would be Joined to the Likes table through the Person2 on Friends table and PersonId on Likes table. Finally, the Likes table would be joined to the Movies table through the MovieId on the Likes table to the movie ID on the Movies table.

Also, on the where statement the condition added would be that the ID on the People table would have to be equal to Joe's ID.

## 9.4 Person with more friends

The code for this exercise consists on creating a registry with the “person” node along with “isfriend” relationship, which connects the person nodes. The registry has the name “aGraph”.

Once the registry is created, there is a call to the degree function that makes the calculations to count the number of connections for each person node with any other person node. The YIELD command associates each node with its corresponding result. The RETURN command fetches the final result, and with the ORDER BY DESC command the result is ordered from the highest value to the lowest one. Finally with the LIMIT command set to 1 only the first and highest value is displayed.

call gds.graph.project('aGraph', 'person', {isfriend: {orientation: 'UNDIRECTED'}});

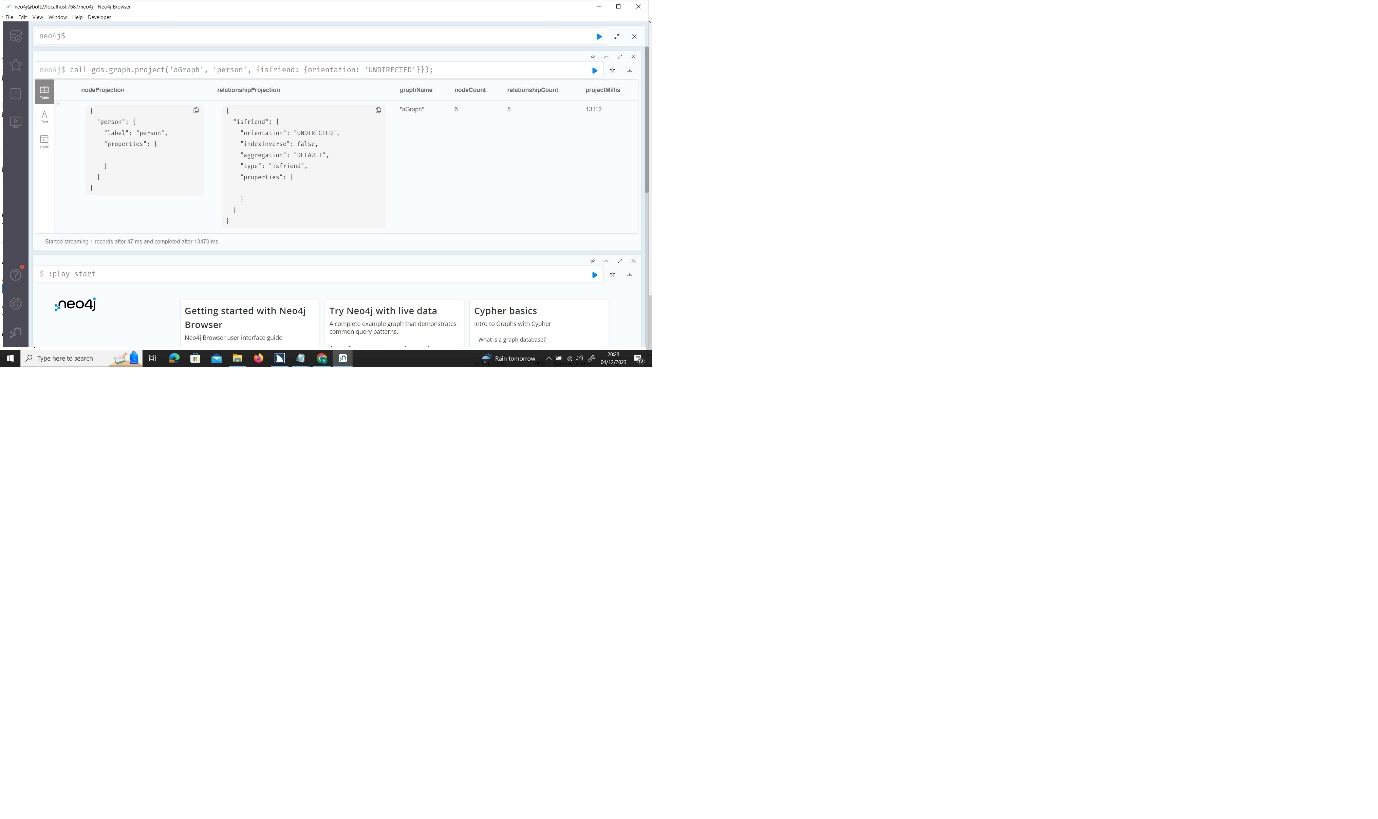


Figure 9.4.a Loading the registry

CALL gds.degree.stream( 'aGraph' )

YIELD nodeId, score

RETURN gds.util.asNode(nodeId).name AS name, score

ORDER BY score DESC

LIMIT 1;

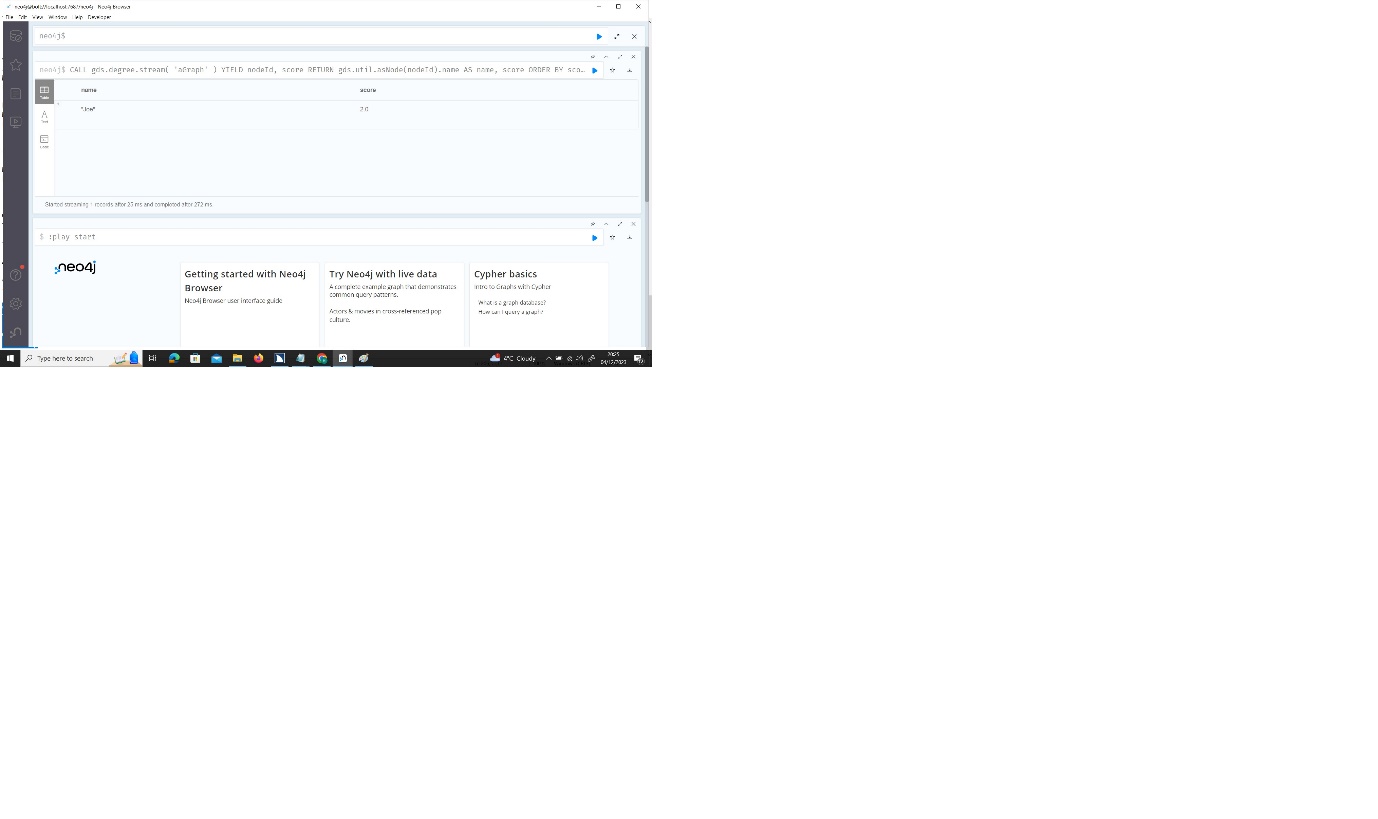


Figure 9.4.b Output

In a relational database, this query would have on the SELECT command the person’s name with the COUNT command with the “\*” parameter to count the results (Alias may be used for a clear display). In the FROM command the person ID from the People table is joined to the Person1 field on the Friends table, and with the OR command to join to the Person2 field, since the required person ID can be in both fields. The GROUP BY with the person ID from the People table would group the results for each person. With the ORDER BY DESC command the results are ordered from the highest value to the lowest one. Finally with the LIMIT command set with 1 only the first and highest result is displayed. The order and the limit commands are very similar whether for the graph or SQL approach.

## 9.5 The Shortest algorithm

MATCH (bruce:Actor {name: 'Bruce Willis'}),

(samuel:Actor {name: 'Samuel L. Jackson'}),

p = shortestPath((bruce)-[:PERFORMS\*]-(samuel))

WHERE all(r IN relationships(p) WHERE r IS NOT NULL)

RETURN p;

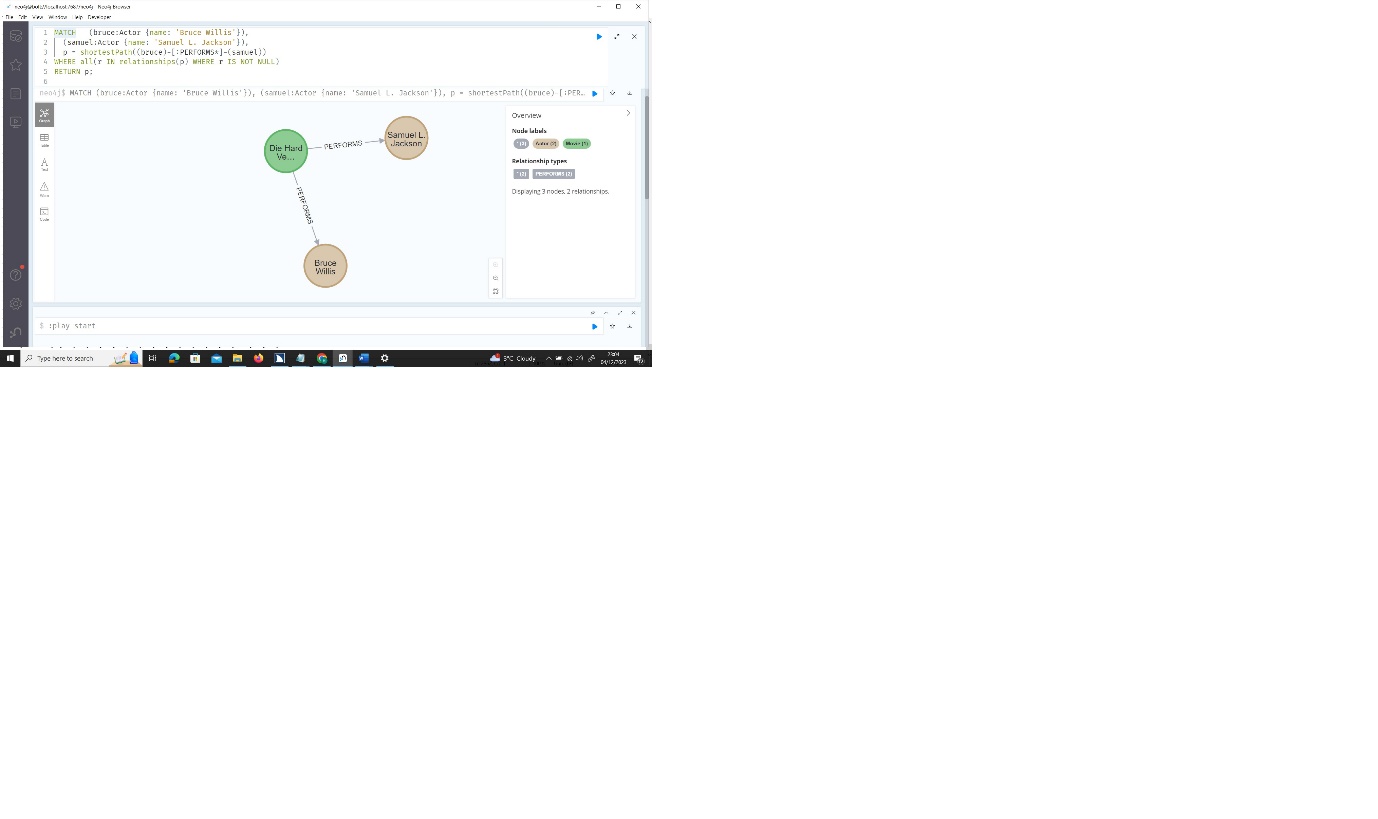


Figure 9.5 Shortest algorithm result

The Shortestpath algorithm finds the common movies between two actors and it displays the path from one to the other.

a new table called Distance is created with 3 fields: the person ID, the movie ID and the distance with the value.

A new row would be inserted with the person ID and movie ID with the distance set to m0 for initialization.

Then a loop would iterate the Distance table looking for the records where the person ID and movie ID are.

Within the loop a SELECT command would iterate all the Likes table and would add 1 value to the distance field in the Distance table on the corresponding record where the person ID and movie ID are.

After the loop a SELECT command that selects the record where the person ID and movie ID are and retrieves the value on the distance field, which is the final result.

a new table called Distance is created with 3 fields: the person ID, the movie ID and the distance with the value.

A new row would be inserted with the person ID and movie ID with the distance set to m0 for initialization.

Then a loop would iterate the Distance table looking for the records where the person ID and movie ID are.

Within the loop a SELECT command would iterate all the Likes table and would add 1 value to the distance field in the Distance table on the corresponding record where the person ID and movie ID are.

After the loop a SELECT command that selects the record where the person ID and movie ID are and retrieves the value on the distance field, which is the final result.

# 12. Comparison and recommendation

## 12.1 Introduction

Relational databases are designed for transactions. These databases focus on storing data without the need of making very complex analyses, just to record the data. Relational databases store data on tables. Tables have an information structure which is common for all the data stored on the tables (Mengraph, 2023).

The structure of tables is composed of columns, with each column having a unique name. is Each column is designed to store a specific value/attribute of data and may contain data of different of formats. Rows or tuples are the part of the table that represent the set of values of that record. Rows are composed of the columns of the table.

On relational databases, rows can have primary key, a unique value on one or composed column for each row that identifies that row in all the table. Also, rows can have foreign keys. Foreign keys are the value of a column that is shared with a primary key on another table. The key ensures the integrity of the whole database. Foreign keys are used for connecting tables.

Graph databases store Data in nodes, which correspond to the rows on relational databases. Nodes are connected in the form of relationships. Both nodes and relationships contain attributes/properties that define each of these components of a graph database. The Structure of the data is not fixed; therefore, nodes and relationships may have different data structures.

Both nodes and relationships contain labels. Labels identify the group either the nodes or relationships belong to.

Graph databases have a more friendly representation for human beings, with nodes and connections among them. This is more helpful in a context of making decisions after interpretating the data representation. Graph databases use Cypher query language that is more oriented on querying relationships.

Relational databases use Structured Query Language (SQL) for working with data, and focus more on tuples and attributes

## 12.2 Functionality

Relational databases are suitable for a well-structured predefined data and where it is not expected that the structure suffers too many changes. The core of this type of database is the data and not the relationships among all the tuples. Relational databases require too many resources to allow the user to make a query where a large number of tables are used.

Graph databases are designed to manage data that is very interconnected among the nodes and are not focused on data. These databases have a large number of algorithms that facilitate analysis of the relationships. Graph databases do not have a fixed and predefined data structure. Therefore, integrity is not guaranteed on these databases.

Graph databases do not have a universal standard for querying data. Different database management systems have different query language. Therefore, programmers cannot switch graph databases as easy as relational databases (Graph Database Market Size, Industry Share, Emerging Trends & Opportunities | MarketsandMarkets™, 2023).

Relational databases use SQL as standard for working data. Therefore, switching from one DB engine to another is easy for programmers.

## 12.3 Market shared

Relational databases are used in sectors where integrity and consistency are key to the data storage. The most important factor is to maintain a database where data cannot be corrupted and transactions are complete only if all the operations involved are performed successfully. There is not a deep analysis of the relationships among data elements. The most important feature is to have a reliable and consistent data storage. Some examples are: financial, health system, public administration or sectors where people’s data integrity is crucial (business customers, human resources…).

Graph databases are used in sectors where the relationships among data is the most important feature. Where data is constantly changing and relationships are changing too and have their own attributes. The analyses made with data is based on the connections among the nodes. These analyses try to find the pattern that explain the reason of the relationships and reach a conclusion. Some examples are: social media, recommendations for customers according to their behaviours or geolocation. Where the connections among the data is more relevant than the data itself.

## 12.4 Scalability

Relational databases are not designed for a very complex data storage. In the case of having a large number of relationships, relational databases become less efficient and more resources are needed. Moreover, these databases are designed for mostly working on one machine with no distributed data. In recent years with cloud storage, data is expected to be save in different platforms. Relational databases have improved in this sense, but the resources needed are high (Allen, 2023).

Graph databases, as a non-SQL database version, is designed to work with distributed data and cross-platforms. These databases do not have the need of using too much resources for working with data, therefore they are more affordable. Moreover, graph database can handle large multimedia files.

## 12.5 Flexibility

Relational databases have a strict data schema. Data structure can be modified, but the entire table is affected by the change. These databases may accept null values, in the case of the record does not contain data for that column. SQL can make queries on run time and can use views that provide information without changing the source. Relational databases can use triggers or procedures in order to assist with any operations made on the database (Smallcombe, 2023).

Graph databases are flexible, since data structure can be different among all nodes and relationships. Each node or relationship can contain different information to the other components. Nodes and relationships can even change the data structure without affecting the database. New relationships can be added dynamically regardless of the node data schema (Smallcombe, 2023).

## 12.6 Recommendation

For this project the option chosen as the most optimal in order to manage all the information through the best way is the hybrid option.

The section that handles the data of movies, actors, genres and their relationships must use relational databases. The reason of this is that this data has a predefined structure and it is unlikely to suffer too many changes. The data that both movies and actors must have is very clear and must be fixed for all rows in order to maintain consistency on the database. Also, there is a low likelihood that data will have any change in the future. In the case of the data suffers a change will not be often and will not have a big impact on the database.

The relationships are also well predefined and there is a need to guarantee that the connections maintain the integrity on the database, because the project must ensure that an existing movie is linked to an existing actor, and the same for the genres. This is guaranteed by primary keys and foreign keys. Also, the number of queries on the relationships is not high, since the queries are needed just to list the movies with actors, and this operation is not very often. Nor are there big algorithms involved for analysing the relationships on the database, therefore, there will not be a high CPU usage for running algorithms.

The section that handles the social media: people, friendships and likes must use graph databases. The reason of this is that the core of this section is the different and numerous relationships that are involved on this part of the project. Also, one of the keys of this section of the solution is to analyse with big algorithms the reason and consequences of those connections. Graph databases handle the query of a large number and complex relationships and the implementation of the algorithms more efficiently than relational databases.

The data itself is not so relevant and the data schema may suffer changes and the databases will not have a big impact on its results. Therefore, there is no need for a strict data schema nor primary keys and foreign keys to maintain a high level of integrity on the database.

# References

Allen, M. (2023, 11 27). *Relational Databases Are Not Designed For Scale - MarkLogic.* Retrieved from marklogic.com: https://www.marklogic.com/blog/relational-databases-scale/

*Graph Database Market Size, Industry Share, Emerging Trends & Opportunities | MarketsandMarkets™*. (2023, 11 27). Retrieved from marketsandmarkets.com: https://www.marketsandmarkets.com/Market-Reports/graph-database-market-126230231.html

Mengraph. (2023, 11 27). *Graph Database vs Relational Database.* Retrieved from memgraph.com: https://memgraph.com/blog/graph-database-vs-relational-database

Neo4J. (2023, 11 10). *Modeling: relational to graph - Getting Started*. Retrieved from neo4j.com: https://neo4j.com/docs/getting-started/data-modeling/relational-to-graph-modeling/

Neo4J. (2023, 11 10). *Transition from relational to graph database - Getting Started*. Retrieved from neo4j.com: https://neo4j.com/docs/getting-started/appendix/graphdb-concepts/graphdb-vs-rdbms/

Smallcombe, M. (2023, 11 27). *SQL vs NoSQL: 5 Critical Differences | Integrate.io.* Retrieved from integrate.io: https://www.integrate.io/blog/the-sql-vs-nosql-difference/#:~:text=Non%2Drelational%2C%20or%20NoSQL%20databases,rows%2C%20tables%2C%20or%20keys.