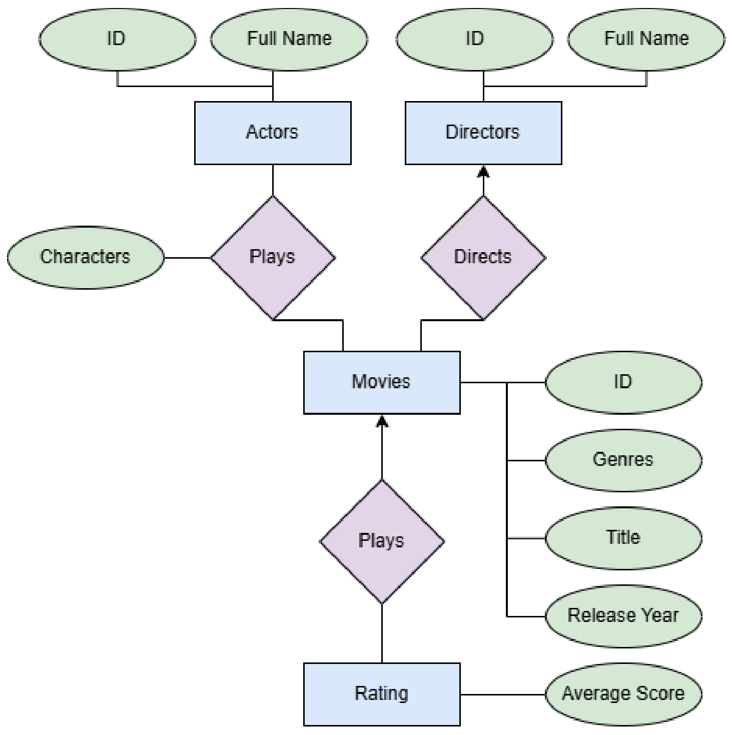
System Documentation

Oz Cabiri & Assaf Yaron

ER Diagram:



Database Schema:

Entities
Actors(actor_id, name)
Directors(director_id, name)
Movies(movie_id, title, start_year, genres)

Movies-Actors(movie_id, actor_id, characters)
 	
FK(movie_id → Movies.movie_id)
FK(actor_id → Actors.actor_id)

Movies-Directors(movie_id, director_id)
 	
FK(movie_id → Movies.movie_id)
FK(director_id → Directors.director_id)

Ratings(movie_id, average_rating)
 	
FK(movie_id → Movies.movie_id)

Database table reasoning:

The data was retrieved from IMDB’s Non-Commercial Datasets. Some of the design decisions were followed directly from IMDB’s implementation. The rest were derived from the queries we decided to implement and the attempt to optimize them.

All of our queries regard actors and data that can be learned about them. Since only 1 query needs directors’ names, we decided to split them into 2 tables: *Actors* and *Directors*. This can create more efficient indices (for example: full-text index on actors names only), as well as keep tables shorter, which results in more efficient joins.

This is also why we chose to not include a ‘parent’ *Persons* table: the queries do not require nor benefit from such a table and discarding it reduces storage needs.

Unintuitively, we have decided to not add a *director* column nor a *ratings* column to the *Movies* table. While, the relations *Movies-Directors* and *Movies-Ratings* is many-to-1 (as shown in the ER diagram), adding these columns will tie them to *Movies* and require us to use it for nearly every query, even though only 2 queries use the table. The added cost of 2 tables (which realistically, only add 2 columns) is a reasonable trade-off in our opinion.

Actors and *Directors* contain a name field which holds the full name of a person. There was no need to split the field in *Directors* because we don’t join or filter based on that attribute, and splitting it in *Actors* was unnecessary as well, since we wanted to allow a more flexible search on actors’ names (hence the use of full-text). An added bonus is the reduced workload in processing the data from IMDB’s datasets – we avoid splitting each entry to first and last name, as well as middle name which might contain null, thus keeping the database ‘cleaner’ and less prone to errors.

Regarding *genres* under *Movies*, we decided not to use separate tables, but instead maintain this data as lists.

Since *genres* are only used for 1 query and have a many-to-many relation to *Movies*, we decided it would be more space efficient to keep them as lists for each movie, rather than a separate table and considering the data from IMDB was already in list form it was also simpler to implement.

Database indices reasoning:

For the full-text queries we have a full-text for each, since they operate on different fields and contain unrelated data. They use NATURAL LANGUAGE MODE to ignore characters such as ‘ “ ‘ and ‘ , ‘.

Additionally, we added indices on the foreign keys in *Movies-Actors* and *Movies-Directors* to create more efficient joins on queries 4 and 5.

An index was created on *Ratings.average_rating* to improve efficiency in query 6.

Queries description:

1. Find movies with a certain genre. The query uses full-text search on *Movies.genres*, which we build an index for in the database creation. The query uses prepared statements to prevent SQL injection.
2. Find actors who played a certain character. The query uses full-text search on *Movies-Actors.characters*, which we build an index for in the database creation. The query uses prepared statements to prevent SQL injection.
3. Find a movie with a certain actor by the actor's name. The query uses full-text search on *Actors.name*, which we build an index for in the database creation. The query uses prepared statements to prevent SQL injection.
4. Find the top N actors with the most movie appearances. The query receives a number N using prepared statements to prevent SQL injection. The index we added on *Movies-Actors.actor_id* help with join efficiency.
5. Find the top N actor-directors collaborations. The query receives a number N using prepared statements to prevent SQL injection. The indices we added on *Movies-Actors.actor_id, Movies-Actors.movie_id, Movies-Directors.director_id, Movies-directors.movie_id* help with join efficiency.
6. Find actors who played in above-average rated movies. The query calculates the average rating for all movies and then chooses rows that exceed this value.
   Like in the previous queries the indices on *Movies-Actors.actor_id, Movies-Actors.movie_id* help with a more efficient join. Additionally, the index on *Ratings.average_rating* improves efficiency on selecting above a certain value (in this case, the average rating of all movies).

Code structure and API usage:
The code uses csv already present in the data folder. These were obtained from IMDB’s Non-commercial Datasets and filtered with another script – these are not included due to file size and relevancy.

Step 1: run *create_db_script.py*, which connects to the server and creates a *mysql.connector.connect* object, which is used to create the tables and indices.
Each table is created as described in the database schema and is wrapped in try-except for more granular error detection and future handling. The same applies for each index.

Step 2: run *api_data_retrieve.py*, which connects to the server and creates a *mysql.connector.connect* object, which is used to insert data into each table using a method for each. For *genres* and *characters* we also parse the json arrays to remove unwanted characters.

Step 3: run *queries_execution.py*, which uses queries from *queries_db_script.py* as examples.

Step 4: *queries_db_script.py* implements the queries, each wrapped in try-except for graceful error detection and future handling.