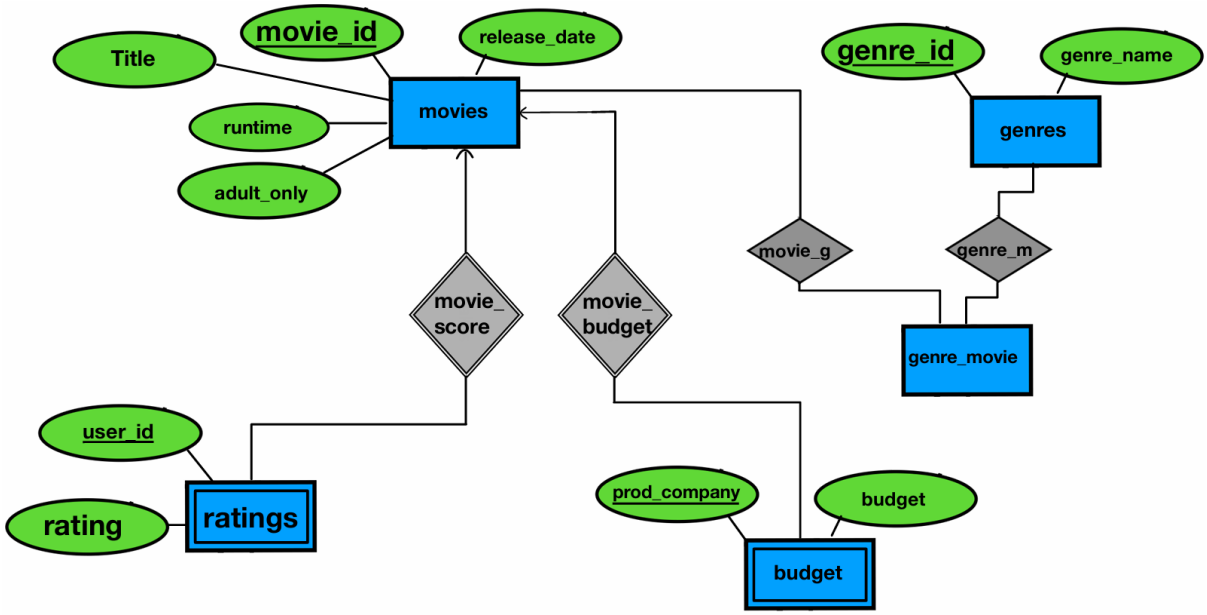
Software Documentation

This application provides information regarding movies.
The database schema is designed to efficiently store and retrieve data, including information about movies, ratings, budget, genres, and production companies.



The schema is built from five tables:

Relations:

- movies(movie_id, title, runtime, adult_only, release_date)
- genres(genre_id, genre_name)
- ratings(movie_id, user_id, rating)
- budget(movie_id, prod_company, budget)
- genre_movie(movie_id, genre_id)

Foreign Keys:

- ratings: movie_id -> movies: movie_id
- budget: movie_id -> movies: movie_id
- genre_movie: movie_id -> movies: movie_id
- genre_movie: genre_id -> genres: genre_id

Reasoning for the chosen database design

Considering the relation between movies, genre_movie and genres: This setup allows for flexibility in associating movies with multiple genres and genres with multiple movies without duplicating data or creating complex relationships directly between the movies and genres tables.

The chosen design prioritizes normalization, which offers several advantages:

- Efficiency: With properly indexed foreign keys, joins can be performed efficiently to retrieve data from multiple tables.

- Reduced Redundancy: Storing information like genre names and production companies in separate tables reduces repetition in the movies table.

- Improved Data Integrity: Changes to a genre name needs to be made only in one table. This reduces the risk of errors.

Drawbacks of Alternative Designs
Another option could involve adopting a more denormalized structure wherein genre names, budget, and production company are directly incorporated into the movies table. While this approach may simplify certain queries by eliminating the appliance of joins, it would notably heighten data redundancy and the likelihood of inconsistencies.

Explain database optimizations, including index usage.

- Indexes are used to improve the runtime and performance of queries.
- As required, full-text indexes in movie and genre tables over the title column and genre_name column. Those indexes allow us to perform the first and second queries more efficiently.
- Primary keys of genre_movie which is made of two primary keys from movies and genres improves join and lookup performance.

The main queries:

Each query that requires an input from the user handles SQL injection threat because of the following reasons:

- The SQL query is constructed as a string with a placeholder %s for the input parameter.
- When executing the query the input value is provided as a parameter in a tuple. This ensures that the input value is treated as data and not as part of the SQL command.

`query_1` performs a full-text search for movie titles based on a specific word provided as input. Using the MySQL `MATCH ... AGAINST` syntax for natural language full-text searches.

Parameters:
`word`: A string representing the word to search for in movie titles.

The query results are fetched and printed to the console.

Running example
```
query_1("love")
```
This will search for movie titles containing the word "love" and print the results to the console.

`query_2` counts the number of movies in each genre according to predetermined budget buckets. It categorizes movies into "Low Budget," "Medium Budget," or "High Budget" based on their budget and counts the movies within each genre and budget bucket.

This query has no parameters.

Running example
```
query_2()
```
This will print the count of movies in each genre categorized by their budget buckets.

`query_3` lists movies along with their average rating given by all users.

This query has no parameters.

Running example
```
query_3()
```
This will print the list of movies along with their average rating.

`query_4` retrieves the names of the movies rated by a specific user along with their ratings.

Parameters:
`user_id`: An integer representing the ID of the user whose ratings are to be retrieved.

Running example
```
query_4(123)
```
This will print the list of movies rated by the user ID 123 along with their ratings.

`query_5` retrieves details about a movie based on its title. It returns the movie's runtime, average rating, and the number of voters who rated the movie.

Parameters:
`movie_title`: A string representing the title of the movie for which details are to be retrieved.

Running example
```
query_5("Avatar")
```
This will print the following details about the movie "Avatar", including the runtime, average rating, and number of voters.

`query_6` retrieves movies released in a specified year along with their budgets. It orders the movies by their budget in descending order.

Parameters:

`year`: An integer representing the year for which movies are to be retrieved.

Running example
```
query_6(2009)
```
This will print the list of movies released in 2006 along with their budgets, ordered by budget in descending order.

`query_7` retrieves for every production company, the movie title with the highest budget. It returns a descending list by budget of movie titles along with their respective production companies, and budgets.

This query has no parameters.

Running example
```
query_7()
```
This will print the list of movie titles with the highest budget for every production company.

Outline code structure, CSV usage, and the general flow of the application.
Outline code structure:
We create various tables, insert the data and imply various query functions.

CSV usage:
Data insertion is performed by reading and parsing data from CSV files, that were pre-normalized and well separated. This step is prior to the database insertion.

General flow:

- Data Preparation: Read the original data from CSV files, for the corresponding tables mentioned above.
- Database Setup: Establish a connection to the database and construct the essential tables as per the schema, ensuring configuration of all indexes and full-text indexes.
- Data Insertion: Populate the database with data extracted from the preprocessed CSV files, inserting them into the respective tables.
- Query Execution: Execute queries, either prompted by user input or predefined operations.