Final report *MovieMate*

Songyang Du, Shengyin Si, Yahan Xu, Yu Yang

Abstract

This report is designated for the CIS550 Database Design course project in Spring 2023. The project focuses on several aspects of database design, including schema design, normalization, index and query optimization. The final application is a fully functional website that utilizes a database system capable of supporting the platform's data management needs. The report outlines the development process, the design decisions made, and the lessons learned throughout the project.

Keywords: Database Design; Web Development; Data Scraping; Query Optimization; Divide and Conquer

The MovieMate provides a simple and user-friendly solution for movie enthusiasts who struggle to find relevant and accurate information. MovieMate allows fuzzy search and full-text search, which improves flexibility. Additionally, this application serves as a solid starting point for further development, such as implementing payment options and personalized recommendation systems. Throughout the development process, with features and theories covered in the class, methods for improving performance through the utilization of normalization and query optimization have been explored.

Introduction

There is an increasing demand for online movie databases as more people shift towards streaming online. However, many existing databases are often cluttered and difficult to navigate, making it challenging for users to find relevant and accurate information. This project aims to provide a user-friendly solution by allowing them to easily search and browse through a vast collection of movies. By including complex database queries into the application cross all the three datasets, eg: movies, actors, and Academy rewards, users can gain insights that are not easily available elsewhere or in a more efficient way. For example, users could utilize the web app to determine which movies were trending at certain dates and in certain countries. In addition, MovieMate could also be a valuable source of information for researching movie-related topics such as actors, directors, and Academy Awards.

Technology Stack and System Architecture

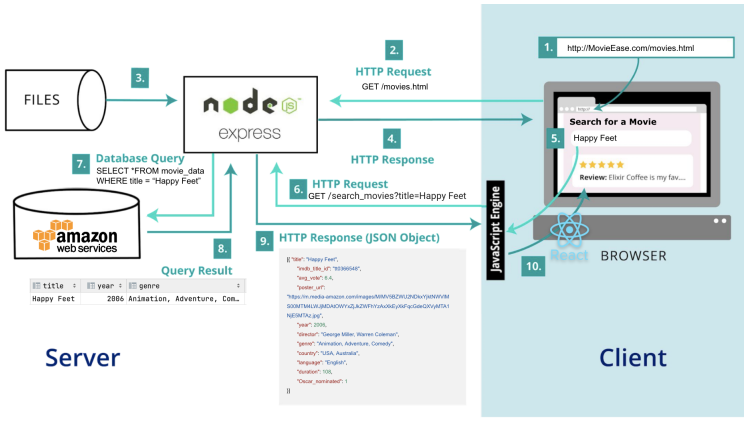
list of technologies

- **Python(Jupyter Notebook)**: data cleansing and data scraping
- **MySql**: managing the data
- **Node.js**: running environment for JavaScript
- **React**: building interactive user interface
- **CSS**: styling and formatting web pages
- **HTML**: creating structures and contents of the webpage
- **MUL**: building visually appealing web applications
- **Recharts**: create interactive and responsive charts and graphs
- **Git**: version control
- **Postman**: API testing
- **Midjourney**: generating logo
- **Balsamic Mockup**: interface design tool for creating wireframes
- **Premiere Pro**: demo editing

System Architecture

There are three core components in this simple web-based application:

- Web Browser: client-side component that interacts with the user by receiving user input, managing the display logic, and validating user inputs if necessary.
- Web Server: server-side component handles all programming logic and processes user requests by sending them to the appropriate component, as well as managing overall application operations.
- Database Server: responsible for all data-related tasks, such as storing and managing the required information for the application. It helps the web application maintain functionalities like searching, filtering, and sorting information based on user requests, as well as presenting the required information to the end-user.



Overview of Data Sources

The three datasets that were selected to be maintain the functionalities of the web application were: Oscar, Moves, and People.

1. Movies: http://www.kaggle.com/datasets/neha1703/movie-genre-from-its-poster kaggle; Owner: Neha
    - Format: csv
    - File size: 9.13MB
    - Total Rows: 40109
    - Total Tables/Relations/csv files: 1
    - Unique movie titles: 39470
    - Column: imdbid, imdb link, title, IMDB score, genre, poster URL
2. Actor: http://www.kaggle.com/datasets/trentpark/imdb-data kaggle; Owner: Trent Park
    - Format: csv
    - File size: 47.93MB
    - Total Rows: at least 83,000 rows each file, name.csv has 297,711 rows
    - Total Tables/Relations/csv files: 4 ie: movies.csv, name.csv(used), ratings.csv, title principals.csv
    - Column: imdb name id, name, birth name, height, bio, birth detials, birth date, place of birth, death details, date of death, place of death, reason of death, spouses string, spouses, divorces, spouses with children, children
3. Oscar Award: https://www.kaggle.com/datasets/unanimad/the-oscar-award kaggle; Owner: The Devastator
    - Format: csv
    - File size: 948.21 kB
    - Total Rows: 9856
    - Total Tables/Relations/csv files: 1
    - Column: year film, year ceremony, ceremony, category, name, film, winner
4. Actor poster (Web scraping) Using python 3 imdb module to scarping imdb.com to get the image of the actor.

Database

Data Ingestion Procedure

- Remove duplicates and redundant columns.
- Filtered out invalid data such as missing names for movies and people. Used year, movie title, and movie original title as the merging key to attach Imdb_title_id from the Movie table to the Oscar database.
- Ensured consistency between movie titles in Oscar and Movie tables. Leveraged the Python IMDb package to extract people's profile photo URLs from IMDb.
- Utilized concurrent.futures to improve the automation and efficiency of the data retrieval process.
- Created an additional column, Photo_url, in the People table to store profile photo URLs. Formatted date of birth and death in the People table, using only the year in case of incomplete dates of birth.

These techniques significantly improved the quality and usability of our data, making it more accurate, efficient, and streamlined.

DDL and Entity Relationship Diagram

See appendix 1.1 and 2.1.

Normalization and Entity Resolution

By definition, a relation is in 3NF if for every $X \to A$ that holds over R, either:

- either trivial $A \in X$, (Req.1.1)
- or X is a superkey for R, (Req.1.2)
- or A is a prime attribute (part of some keys), (Req.1.3)

By definition, a relation is in BCNF if for every $X \to A$ that holds over R, either:

- either trivial $A \in X$, (Req.2.1)
- or X is a superkey for R, (Req.2.2)

All of the four tables are either in 3NF or BCNF since:

1. Movies table(15,200 rows):imdb_title_id (primary key), title, original_title, year, genre, duration, country, language, description, avg_vote, votes.
    - primary key: imdb_title_id
    - imdb_title_id $\to$ title
    - imdb_title_id $\to$ year
    - imdb_title_id $\to$ genre
    - imdb_title_id $\to$ duration
    - imdb_title_id $\to$ country
    - imdb_title_id $\to$ language
    - imdb_title_id $\to$ description
    - imdb_title_id $\to$ avg_vote
    - imdb_title_id $\to$ votes
    - imdb_title_id $\to$ Oscar_nominated
    - imdb_title_id $\to$ poster_url
    - imdb_title_id $\to$ director

    The Imdb title id is the only candidate key as multiple films can have the same title/original title. Similarly, there can be many movies released in a single year, and this applies to other attributes like country and language. During the data processing step, it was verified that there were no duplicate Imdb title ids, and the description attribute contained null values which means description can no be primary key.

    Furthermore, it is evident that each function dependency stated satisfies the requirements of Req.1.1 and Req2.2. As a result, this relation is in 3NF/BCNF, and all attributes are critical to the website's functionality.
2. movie_people(150,103 rows): imdb_title_id, ordering, imdb_name_id, category
    - the set of primary key are: imdb_title_id, ordering
    - imdb_id, ordering $\to$ imdb_name_id
    - imdb_id, ordering $\to$ category

    An additional attribute ordering has been added to assign an order to each person based on their jobs. For instance, a person can have multiple indexes if they have worked in various roles in a movie.
3. People(67,267): imdb_name_id, name, birth_name, date_of_birth, height, place_of_birth, date_of_death, bio, photo_url

- the set of primary keys are: imdb_name_id
- imdb_name_id → birth_name
- imdb_name_id → date_of_birth
- imdb_name_id → height
- imdb_name_id → place_of_birth
- imdb_name_id → date_of_death
- imdb_name_id → bio
- imdb_name_id → photo_url

Similarly, it is evident that each function dependency stated satisfies the requirements of Req.1.1 and Req2.2. As a result, this relation is in 3NF/BCNF, and all attributes are critical to the website's functionality.

4. Oscar(5,970): year_film, year_ceremony, category, name, winner, imdb_title_id, movie_title

   - the set of primary keys are:ceremony(CEY), category(CTY), name(N), movie_title(MT)
   - CEY, CTY, N, MT→ year_film
   - CEY, CTY, N, MT→ year_ceremony
   - CEY, CTY, N, MT→ winner
   - CEY, CTY, N, MT→ imdb_title_id

   Four attributes have been chosen as primary keys, since one movie can be nominated for different categories, and the movie name might be duplicated. Overall, it is evident that each function dependency stated satisfies the requirements of Req.1.1 and Req.2.2. As a result, this relation is in 3NF/BCNF, and all attributes are critical to the website's functionality.

   Resolution efforts have been made to ensure that each movie in the database is correctly associated with the people who worked on it, and that there are no duplicate entries. Additionally, each Oscar-nominated movie is accurately linked to the relevant award categories and nominees.

Description of Website Application and Features

- Homepage:
  1. Daily recommendation: one randomly generated Oscar-winning movie
  2. List the top 50 recent movies for selected genre
  3. List the top 50 rated movies for selected language
  4. When hovering the mouse over a movie's poster, a brief description of the film is displayed.
- Movies main page:
  1. Display all movie cards with title and rating
  2. Customer filter/searching function: title, year, genre, language, country, isOscar
- Movies details page:
  1. Display the movie's basic information, poster, brief description and related people
  2. The people who participate in the movie will be below the movie card. By clicking people's Avatar, the web page will be directed to its details page
- People's main page
  1. List of profile photo for every actor, actresses, and directors
  2. Allows users to search people by name
- People's detail page
  1. Display people's basic information, poster, brief description, and related movies
  2. Display related movies: by clicking the movie poster, the web page will be directed to it's details page
  3. Display related actors: by clicking people's profile photo, the web page will be directed to it's details page
- Oscar main page
  1. Shuffle slides: display some photos with a short description
  2. Users can search the Oscar nomination information list for a certain year, including Year, category, Title, Name, Rating, whether Win
  3. By clicking the movie title, the web page will be directed to the it's details page
  4. Shuffle slides: display some photos for the first woman to win in some of the non-gendered Oscar category
- Hall of Fame
  1. Displays top 10 Oscar winning directors and their number Oscar nominations and wins. By clicking the director profile photo, the web page will be directed to their details page.
  2. Displays statistics of legendary actress and various statistics about their careers. By clicking the actress profile photo, the web page will be directed to their details page.
  3. Displays actors and actresses who have received the most Oscar nominations in each decade.

API Specification

1. Route: /movies
   - description: return all movies, ordered by year
   - route parameters: None
   - query parameters: None
   - route handler: movies(req, res)
   - return type: JSON Array
   - return parameters: {{imdb_title_id (string), title (string), avg_vote (float), poster_url (string) }}
2. Route: /movies/:movie_id
   - description: return return all the information about a movie with given imdb_title_id
   - route parameters: movie_id(string)
   - query parameters: None
   - route handler: movies(req, res)
   - return type: JSON Array
   - return parameters:{imdb_name_id (string), name (string), birth_name (string), date_of_birth (string), place_of_birth (string), date_of_death (string), height float, bio (string) photo_url (string)}, ...}
3. Route: /search_movies

- description: return all movies with given criteria
- route parameters: None
- query parameters: title(string), yearlow(int), yearhigh(int), country(string), language(string), genre(string), isOscard(boolean), page(int), pageSize(int)(default: 10)
- route handler: search_movies(req, res)
- return type: JSON Array
- return parameters: {{imdb_title_id (string), title(string), year(int), genre(string), duration(int), country(string), language(string), director(string), avg_vote (float), poster_url(string), Oscar_nominated (boolean)}, ...}

4. Route: /movie_people/:movie_id

- description: return a list actors in the given movies
- route parameters: movie_id
- query parameters: page(int), pageSize(int)(default: 10)
- route handler: movie_people(req, res)
- return type: JSON Array
- return parameters:name(string),imdb_name_id(string), photo_url(string)

5. Route: /oscar_recommend

- description: return a random Oscar winning movie with the movie title, release year, duration, poster, description, name of the director, and number of oscar nominations
- route parameters: None
- query parameters: None
- route handler: oscarMovieRecommended(req,res)
- return type: JSON Array
- return Parameters:{title (string), year (int), duration (int), imdb_title_id(string), director_name (string), description (string), poster_url (string), num_nominations (int)}

6. Route: /recentgenre/:genre

- description: return the most recent 50 movies with their title, poster and description in the given genre
- route parameters: genre(string)
- query parameters: None
- route handler: recentgenre(req,res)
- return type: JSON Array
- return Parameters:{imdb_title_id(string), title (string), description (string), poster_url (string)}

7. Route: /toplanguage/:language

- description: return the top 50 rated movies with their title, poster and description in the given language
- route parameters: language(string)
- query parameters: None
- route handler: top10language(req,res)
- return type: JSON Array
- return Parameters:{title (string), imdb_title_url(string) description (string), poster_url (string)}

8. Route: /people

- description: return all people
- route parameters: None
- query parameters: page(int), pageSize(int)(default: 10)
- route handler: people(req, res)
- return type: JSON Array
- return parameters {imdb_name_id (string), name (string), date_of_birth (string), photo_url (string), movies_actedIn(int)}

9. Route: /people/:person_id

- description: return all information of given people
- route parameters: person_id
- query parameters: None
- route handler: people(req, res)
- return type: JSON Array
- return parameters: {imdb_name_id (string), name (string), photo_url (string)}

10. Route: /movie_people_acted/:person_id

- description: return highest rated 20 movies of the given actor.
- route parameters: person_id
- query parameters: None
- route handler: movie_people_acted(req, res)
- return type: JSON Array
- return parameters: {title (string), poster_url (string), imdb_title_id (string)}

11. Route: /search_people

- description: search people by name
- route parameters: None
- query parameters: None
- route handler: search_people(req, res)
- return type: JSON Array
- return parameters: {imdb_name_id (string), name (string), photo_url (string)}

12. Route: /avg_vote_person/:person_id

- description: return the average rating of the actor's movies
- route parameters: person_id
- query parameters: None
- route handler: people(req, res)
- return type: JSON Array
- return parameters: {avg_rating}

13. Route: /related_actors/:name

- description: Most related people of the given actor.
- route parameters: name
- query parameters: page(int), pageSize(int)(default: 10)
- route handler: related_actors(req,res)
- return parameters:{name(string),id(string),photo_url(string)}
- return type: JSON Array

14. Route: /search_won

- description: return a list of movie with nomination and winning in the given year.
- route parameters: None
- query parameters: None
- route handler: search_won(req, res)
- return type: JSON Array

- return parameters:{imdb_title_id(string),title(string), country(string),year_ceremony(int), genre(string),avg_vote(int),category(string), winner(int),name(string)}

15. Route: /top_oscar_director

    - description: description: return top 10 Oscar directors with their number of Best Director wins, Best Director nominations, Best Picture wins, Best Picture nominations and average movie ratings, by number of direction wins
    - route parameters: None
    - query parameters: None
    - route handler: top_oscar_director(req, res)
    - return type: JSON Array
    - return parameters:{imdb_name_id(string), name(string),photo_url(string), num_picture_nominations(int), num_picture_wins(int), num_direction_nominations(int), num_direction_wins (int), avg_rating(int)}

16. Route: /stats

    - description: Find the actors who have appeared in the most movies that have been nominated for an Oscar in each decade, along with the number of nominations their movies have received, and the decade in which the movies were released
    - route parameters: None
    - query parameters: page(int), pageSize(int)(default: 10)
    - route handler: oscar_decade(req, res)
    - return type: JSON Array
    - return parameters:{name(string),num_nominations(int), decade(string),avg_rating(int),imdb_name_id(string)}

17. Route: /oscar_actress

    - description: return a list of actresses who have appeared in Oscar-nominated movies, including their maximum age at Oscar nominations, maximum age during movie performance, average age during movie performance, average rating of Oscar-nominated movies, number of Oscar-nominated movies participated in, and total movie participation. The list is sorted in descending order by maximum age at Oscar nominations.
    - route parameters: None
    - query parameters: page(int), pageSize(int)(default: 10)
    - route handler: search_people(req, res)
    - return type: JSON Array
    - return parameters {imdb_name_id (string), name (string), total_movies(int), photo_url (string), oscar_freq(int), max_oscar_age(int), avg_rating(float), max_age(int), average_age(int)}

Queries

Complex Queries

1. Recommend an Oscar-winning movie (Appendix 3.5) The homepage has a daily recommendation function, with the poster, title, duration, number of Oscar nominations, and a short description. The corresponding SQL query retrieves data from Oscar tables and calculates the number of Oscar nominations for each movie. It then joins the results with the movie_data and movie_people tables to obtain additional movie details and director names. The complexity of this query lies in the use of subqueries, aggregations, and multiple joins.

2. Return related actors(Appendix 3.13). This complex query finds the co-actors of a given person up to three levels of separation. It uses a series of CTEs and joins to find the co-actors, avoiding duplicates, and then combines the results into a single table.

3. Return top Oscar Director. This SQL query retrieves data on directors from the Oscar database including their name, IMDb ID, photo URL, number of nominations and wins in the categories of Picture and Directing, and the average rating of their movies as a director. The query uses several subqueries to join and aggregate data from multiple tables. The complexity of the query arises from the use of conditional expressions and the need to count and aggregate data based on specific conditions, such as filtering for directors and different categories of nominations and wins. (Appendix 3.15).

4. Return most nominated people in each decade. This query uses several subqueries to find the actors and actresses who have received the most Oscar nominations in each decade, along with their average ratings. It involves joining multiple tables and using aggregate functions to calculate the maximum number of nominations received in each decade, and then matching that information with the corresponding actors and actresses. The query also includes the use of CTEs to simplify the logic of the query. (Appendix 3.16)

5. Return legendary Actresses. This query is designed to retrieve detailed information about actresses who have been nominated for Oscars, providing various statistics about their careers. The query employs multiple joins and subqueries to extract data from various tables, including the use of CTEs and aggregate functions to calculate important statistics. These statistics include the number of Oscar-nominated movie participations for each actress, the maximum age at which an actress participated in an Oscar-nominated movie, as well as the maximum and average age at which actresses participated in movies. Additionally, the query calculates the average rating of movies in which each actress has participated.(Appendix 3.17)

   All of these five queries are not only crucial to maintain the functionality of the application but also meet the original motivation of the project.

Performance Evaluation And Query Optimization

Exhibit1:Query Optimization Timings		
Page/Query Name	Before	After
1.People Page - Average Rating	332ms	227ms
2.Movie Information Page - Actors/crews of the movie	146ms	112ms
3.People Information Page - Movies, actors acted in	209ms	179ms
4.Homepage Page - Oscar Daily Recommendation	305ms	141ms
5.Hall of Fame - Top Director	741ms	411ms
6.Hall of Fame - Legendary Actresses	671ms	502ms

- For 1,2,3, and other queries: performance improved by switch from AND to JOIN, eliminating unnecessary columns, minimizing the number of joins required, and pushing selections and projections ahead of joins, which reduced the amount of data processed.
- Improved the Oscar movie recommendation(4) by replacing the CTE with a subquery and counting the number of Oscar nominations in the same query with selecting Oscar winning movies. This optimization led to a faster and more efficient Optimized query, which only requires two joins compared to the old query's four joins.
- By implementing subqueries to perform aggregations before joining tables and including the WHERE statement within the subquery, we minimized the amount of data being joined. This resulted in significantly faster query execution. Furthermore, we utilized the IFNULL function to effectively handle null values and ensure the query did not encounter any errors in case of null values within the database. Overall, these optimization techniques improved the efficiency and robustness of the top Oscar director query.
- For the query related to legendary actresses, we optimized the code by combining some subqueries and minimizing the number of joins, which decreased the overall computation time. To further enhance query performance, we utilized the STR_TO_DATE function to convert the date_of_birth column into a date format to address the data type issue.

Testing

Testing has been carried out throughout the development process. Before implementing routes, all queries were tested in **DataGrip** to ensure that no unhandled edge cases would cause the application to crash. After implementing the routes, **Postman** was employed as a faster and more traceable testing method to verify that the actual outcomes matched the expected ones. Additionally, a group of 3-4 people who were not involved in the development process tested the application, and based on their feedback, the UI and layout were modified to provide a more user-friendly experience.

Technical challenges

1. Data cleaning process:
   - The data scraping process presented significant challenges due to the large amount of data, which was estimated to take over 80 hours to scrape. We encountered issues such as slow responses, random errors, and memory constraints, making the task even more daunting. To overcome these challenges, we implemented a complex partitioning strategy, which involved splitting all people IDs into 100 partitions and utilizing parallel processing to improve efficiency. This approach required a significant amount of technical expertise and resourcefulness, but ultimately proved to be successful in achieving our data scraping goals.
   - The lack of an imdb_title_id in the Oscar table resulted in a discrepancy in the connections between the Oscar and Movie tables. Additionally, the non-uniqueness and variations in different languages of the movie title rendered a single movie name match ineffective. To address these issues, we developed a comprehensive matching method that takes into account the year, title, and original title in the Movie data to match the given Oscar movies. This approach significantly increased the matching rate and accuracy, resulting in a more reliable dataset.
   - One of the key functionalities of the web application is calculating the age of people. However, the date of birth and death in the People table were poorly formatted and contained redundant information. To improve data accuracy, we implemented a formatting process that identifies and standardizes various date formats, converting them into a consistent format. In cases where only the year was available, we retained only the year, streamlining the date of birth format and enhancing the accuracy of our presented query results.
2. Slow performance of query execution.
   - As stated in Performance Evaluation, there is a trade-off between filtering out rows containing null values or keeping them to maintain data integrity. Modified the query to include null checking without slowing down overall performance.
3. Status recognition.
   - Due to the time gap between a user performing a search action and the results being displayed on the screen, the page would immediately render the 'no results found' message before the page loaded. This issue occurred for all search functions. To better handle the status recognition of the search results, improve users experience and make the underlying logic more complete, a React Hook useState(false) was created. This hook helps to determine whether the empty results are due to no matching results or if the data is still loading.
4. React component reuse.
   - To reuse the component as well as maintain individual feature, container component pattern has been employed as many as possible. For example, challenges arose when displaying large amounts of data. In order to maintain UI consistency, save coding time, and

make maintenance easier, the MUI pagination module has been extended with careful design and error handling to support the functionality of the application.

5. Homepage Design.

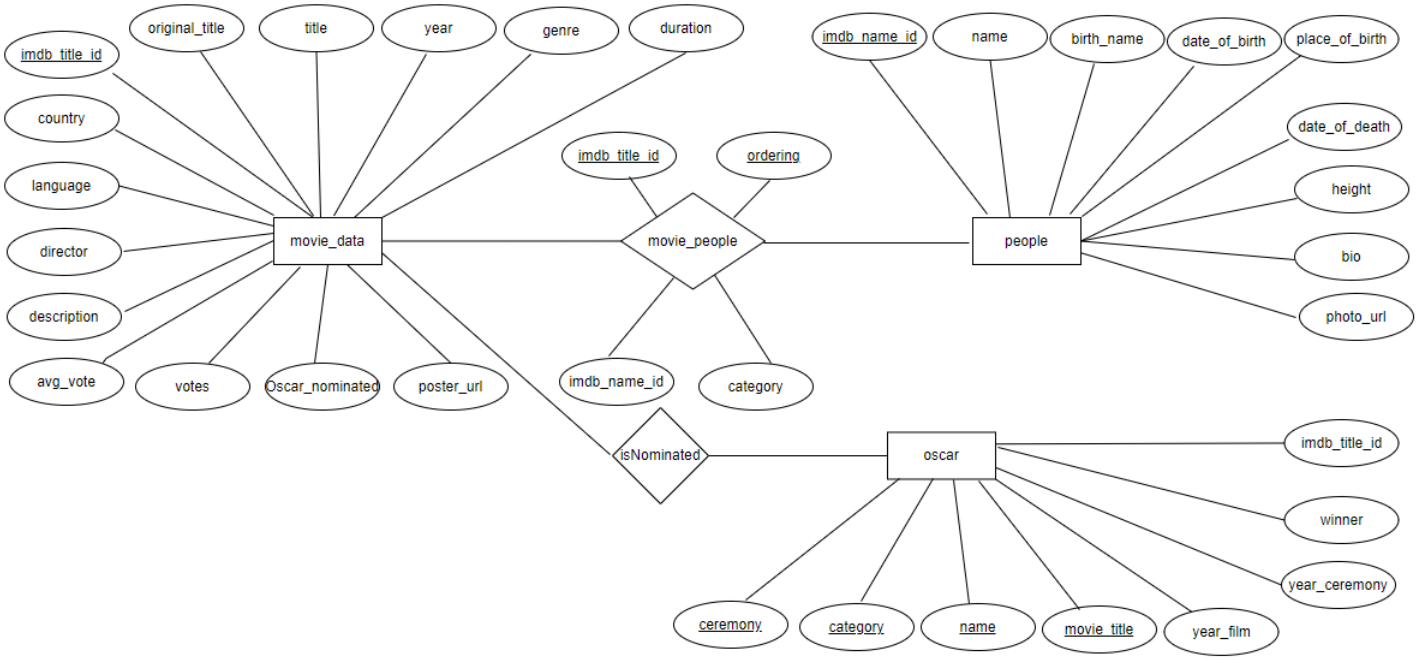
    - Designing the homepage of our web application presented several challenges, mainly due to the complex structure and numerous MUI components involved. In order to achieve our desired web page design, we invested significant effort in mastering the functionalities of various components and incorporated multiple useState hooks. However, the implementation process was frequently hindered by unfamiliar errors and warnings, which caused unexpected behaviors in the application. Successfully navigating these challenges demanded a high degree of technical proficiency and adaptability, and we made extensive use of the browser's inspect functionality to detect and troubleshoot the issues. Although the process was time-consuming and difficult, it ultimately provided a valuable learning opportunity for our team.

Appendix

1 DDL

```
create database MOVIE_DB;
use MOVIE_DB;

create table movie_data
(   imdb_title_id   varchar(20)  primary key,
    original_title  varchar(250) null,
    title           varchar(250) null,
    year            int          null,
    genre           varchar(250) null,
    duration        int          null,
    country         varchar(250) null,
    language        varchar(250) null,
    director        varchar(250) null,
    description     varchar(500) null,
    avg_vote        float        null,
    votes           int          null,
    Oscar_nominated tinyint(1)   null,
    poster_url      varchar(250) null);

create table movie_people
(   imdb_title_id varchar(25) not null,
    ordering      int         not null,
    imdb_name_id  varchar(25) null,
    category      varchar(25) null,
    primary key (imdb_title_id, ordering));

create table oscar
(   year_film     int         null,
    year_ceremony int         null,
    ceremony      int         not null,
    category      varchar(25) not null,
    name          varchar(25) not null,
    winner        tinyint(1)  null,
    imdb_title_id varchar(25) null,
    movie_title   varchar(25) not null,
    primary key (ceremony, category, name, movie_title),
    foreign key (imdb_title_id) references movie_data (imdb_title_id));

create table people
(   imdb_name_id   varchar(25)  not null primary key,
    name           varchar(25)  null,
    birth_name     varchar(25)  null,
    date_of_birth  varchar(25)  null,
    place_of_birth varchar(25)  null,
    date_of_death  varchar(25)  null,
    height         float        null,
    bio            varchar(1000) null,
    photo_url      varchar(150) null);
```

2 ER Diagram



3 Query

3.1 return all movies

```
SELECT title, avg_vote, poster_url,imdb_title_id
FROM movie_data
ORDER BY votes DESC, year DESC;
```

3.2 return all the information of movie

```
SELECT *
FROM movie_data
WHERE imdb_title_id = '${movie_id}'
```

3.3 return a list of movies which met the searching criteria

```
SELECT title, imdb_title_id, avg_vote, poster_url, year, director,
       genre, country, language, duration, Oscar_nominated
FROM  movie_data
WHERE title LIKE '%${title}%'
      AND (${isOscar} IS NULL OR Oscar_nominated = ${isOscar})
      AND year BETWEEN ${yearLow} AND ${yearHigh}
      AND genre LIKE '%${genre}%'
      AND country LIKE '%${country}%'
      AND language LIKE '%${language}%'
ORDER BY votes DESC, year DESC;
```

3.4 return all actors/crews of the given movie

```
SELECT P.name, P.imdb_name_id, P.photo_url
FROM people P
JOIN movie_people MP ON MP.imdb_name_id = P.imdb_name_id
WHERE MP.imdb_title_id = '${movie_id}';
```

3.5 recommend an Oscar winning movie to the user

```
SELECT md.title, md.poster_url, md.description,
       md.imdb_title_id, r.num_nominations,
       md.year, md.duration, p.name AS director_name
FROM ( SELECT imdb_title_id, COUNT(imdb_title_id) AS num_nominations
       FROM oscar
       WHERE winner = true
       GROUP BY imdb_title_id
   ) AS r
       JOIN movie_data AS md
       ON md.imdb_title_id = r.imdb_title_id
       JOIN movie_people AS mp
       ON md.imdb_title_id = mp.imdb_title_id
       AND mp.category = 'director'
       JOIN people AS p
       ON mp.imdb_name_id = p.imdb_name_id
ORDER BY RAND(DATE_FORMAT(NOW(), '%Y-%m-%d'))
LIMIT 1;
```

3.6 return most recent 50 movies with given genre

```
SELECT title, poster_url, description, imdb_title_id
FROM movie_data
WHERE genre Like '%${genre}%'
      AND title IS NOT NULL
      AND poster_url IS NOT NULL
      AND description IS NOT NULL
Order by year DESC, votes DESC
LIMIT 50
```

3.7 top 50 movies with given language

```
SELECT title, poster_url, description, imdb_title_id
FROM movie_data
WHERE language Like '%${language}%'
      AND title IS NOT NULL
      AND poster_url IS NOT NULL
      AND description IS NOT NULL
      Order by avg_vote DESC
LIMIT 50
```

3.8 return all people

```
SELECT p.imdb_name_id,p.name, p.photo_url,
       count(imdb_title_id) movies_actedIn
FROM   movie_people mp join people p on p.imdb_name_id = mp.imdb_name_id
WHERE  mp.category IN ('actor', 'director', 'producer')
group  by p.imdb_name_id
ORDER  BY movies_actedIn DESC;
```

3.9 return all information of given people

```
SELECT *
FROM people
WHERE imdb_name_id = '${person_id}'
```

3.10 movie people acted :person id

```
SELECT M.title, M.poster_url, M.imdb_title_id
FROM movie_people MP join movie_data M on  M.imdb_title_id = MP.imdb_title_id
WHERE MP.imdb_name_id = '${person_id}' and M.avg_vote > 7
order by M.avg_vote DESC
limit 20;
```

3.11 search people by name

```
SELECT *
FROM people
WHERE name LIKE '%${name}%'
ORDER BY name ASC
```

3.12 avg vote person

```
SELECT round(AVG(M.avg_vote),2) as avg_rating
FROM movie_data M, movie_people MP, people P
WHERE M.imdb_title_id = MP.imdb_title_id
AND MP.imdb_name_id = P.imdb_name_id
AND P.imdb_name_id = '${person_id}';
```

3.13 related actors :id

```
WITH X_movies AS(
   SELECT M.imdb_title_id,M.title
   FROM movie_people MP JOIN movie_data M ON M.imdb_title_id = MP.imdb_title_id
        JOIN people P ON MP.imdb_name_id = P.imdb_name_id
   WHERE P.imdb_name_id = '${person_id}'),

coActors1 AS (
   SELECT DISTINCT P.imdb_name_id AS id, P.name AS name, P.photo_url
   FROM movie_people MP JOIN X_movies XM on XM.imdb_title_id = MP.imdb_title_id
        JOIN people P ON P.imdb_name_id = MP.imdb_name_id
   WHERE MP.imdb_name_id != '${person_id}'),

co1movies AS(
    SELECT DISTINCT MP.imdb_title_id
    FROM movie_people MP JOIN coActors1 C1 ON C1.id = MP.imdb_name_id),

coActors2 AS (
   SELECT DISTINCT P.imdb_name_id AS id, P.name AS name, P.photo_url
   FROM movie_people MP JOIN co1movies C1 ON C1.imdb_title_id = MP.imdb_title_id
        JOIN people P ON P.imdb_name_id = MP.imdb_name_id
   WHERE P.imdb_name_id != '${person_id}'
         AND
         MP.imdb_name_id NOT IN (SELECT coActors1.id FROM  coActors1)),

co2movies AS(
    SELECT DISTINCT MP.imdb_title_id
    FROM movie_people MP JOIN coActors2 C2 ON C2.id = MP.imdb_name_id),

coActors3 AS (
   SELECT DISTINCT MP.imdb_name_id AS id, MC.name AS name, MC.photo_url
   FROM movie_people MP JOIN co2movies C2 ON C2.imdb_title_id = MP.imdb_title_id
        JOIN people MC ON MC.imdb_name_id = MP.imdb_name_id
   WHERE
       MC.imdb_name_id != '${person_id}'
       AND
       MC.imdb_name_id NOT IN (SELECT coActors2.id FROM coActors2)
       AND
       MC.imdb_name_id NOT IN (SELECT coActors1.id FROM  coActors1)),
actoerTable1 AS(
   SELECT name, id, photo_url
   FROM coActors1
   UNION
   SELECT name, id, photo_url from coActors2)

SELECT name, id, photo_url
FROM actoerTable1
UNION
SELECT name, id, photo_url from coActors3
LIMIT ${offset}, ${pageSize};
```

3.14 search won

```
SELECT DISTINCT M.imdb_title_id, M.title, M.country, O.year_ceremony as year,
                M.genre, M.avg_vote, O.category, O.winner,O.name
FROM movie_data M JOIN oscar O ON M.imdb_title_id = O.imdb_title_id
ORDER BY O.year_ceremony, O.category;

SELECT DISTINCT M.imdb_title_id, M.title, M.country, O.year_ceremony as year,
                M.genre, M.avg_vote, O.category,O.winner,O.name
FROM movie_data M JOIN oscar O ON M.imdb_title_id = O.imdb_title_id
WHERE (O.year_ceremony = ${year} OR ${year} IS NULL)
ORDER BY O.year_ceremony, O.category;
```

3.15 top Oscar director

```
SELECT p.name, p.imdb_name_id, p.photo_url,
IFNULL(num_picture_nominations, 0) AS num_picture_nominations,
IFNULL(num_picture_wins, 0) AS num_picture_wins,
IFNULL(num_direction_nominations, 0) AS num_direction_nominations,
IFNULL(num_direction_wins, 0) AS num_direction_wins,
ROUND(IFNULL(m.avg_vote, 0), 1) AS avg_rating
FROM
  (
    SELECT
      mp.imdb_name_id,
      COUNT(DISTINCT CASE
        WHEN o.category IN ('Outstanding Picture', 'Outstanding Production',
        'Outstanding Motion Picture',
        'Best Motion Picture', 'Best Picture')
        THEN o.imdb_title_id ELSE NULL END) AS num_picture_nominations,
      COUNT(DISTINCT CASE
        WHEN o.category IN ('Outstanding Picture', 'Outstanding Production',
        'Outstanding Motion Picture',
        'Best Motion Picture', 'Best Picture') AND o.winner = 1
        THEN o.imdb_title_id ELSE NULL END) AS num_picture_wins,
      COUNT(DISTINCT CASE
        WHEN o.category = 'DIRECTING'
        THEN o.imdb_title_id ELSE NULL END) AS num_direction_nominations,
      COUNT(DISTINCT CASE
        WHEN o.category = 'DIRECTING' AND o.winner = 1
        THEN o.imdb_title_id ELSE NULL END) AS num_direction_wins
    FROM
      oscar o
      JOIN movie_people mp ON o.imdb_title_id = mp.imdb_title_id
    WHERE
      mp.category = 'director'
    GROUP BY
      mp.imdb_name_id
  ) AS oscar_data
  JOIN people p ON oscar_data.imdb_name_id = p.imdb_name_id
  LEFT JOIN (
    SELECT
      mp.imdb_name_id,
      AVG(m.avg_vote) AS avg_vote
    FROM
      movie_data m
      JOIN movie_people mp ON m.imdb_title_id = mp.imdb_title_id
    WHERE
      mp.category = 'director'
    GROUP BY
      mp.imdb_name_id
```

```
  ) AS m ON oscar_data.imdb_name_id = m.imdb_name_id
ORDER BY
  num_direction_wins DESC,
  num_direction_nominations DESC,
  num_picture_wins DESC,
  num_picture_nominations DESC,
  avg_rating DESC
LIMIT 10;
```

3.16 oscar decade

```
WITH oscar_movies AS (
    SELECT DISTINCT imdb_title_id, year_ceremony
    FROM oscar
  ),
  oscar_movies_with_rating AS (
    SELECT o.imdb_title_id, o.year_ceremony, m.avg_vote
    FROM movie_data m
    INNER JOIN oscar_movies o
    ON o.imdb_title_id = m.imdb_title_id
  ),
  actor_nominations AS (
    SELECT mp.imdb_name_id,
           CONCAT((om.year_ceremony DIV 10) * 10, '-',
           (om.year_ceremony DIV 10) * 10 + 9) AS decade,
           COUNT(*) AS num_nominations,
           AVG(om.avg_vote) as avg_rating
    FROM movie_people mp
    INNER JOIN oscar_movies_with_rating om
    ON mp.imdb_title_id = om.imdb_title_id
    WHERE mp.category IN ('actor','actress')
    GROUP BY mp.imdb_name_id, decade
  ),
  actor_nominations_max AS (
    SELECT MAX(num_nominations) AS max_nominations, decade
    FROM actor_nominations
    GROUP BY decade
  )
  SELECT p.name, an.max_nominations as num_nominations, an.decade,
         ROUND(a.avg_rating, 1) as avg_rating,a.imdb_name_id
  FROM actor_nominations_max an
  INNER JOIN actor_nominations a ON a.decade = an.decade
             AND a.num_nominations = an.max_nominations
  INNER JOIN people p on a.imdb_name_id = p.imdb_name_id
  WHERE an.decade <> '1920-1929'
  LIMIT ${offset}, ${pageSize};
  `;
```

3.17 oscar actress

```
WITH oscar_movies AS (
    SELECT DISTINCT imdb_title_id, year_ceremony
    FROM oscar
),movie_counts AS (
    SELECT imdb_name_id, COUNT(DISTINCT imdb_title_id) AS total_movies
    FROM movie_people
    WHERE category = 'actress'
    GROUP BY imdb_name_id
),movie_ages AS (
  SELECT mp.imdb_name_id,
         p.name,
         p.photo_url,
         STR_TO_DATE(p.date_of_birth, '%m/%d/%Y') AS date_of_birth,
         MAX(m.year - YEAR(STR_TO_DATE(p.date_of_birth, '%m/%d/%Y'))) AS max_age,
         ROUND(AVG(m.year - YEAR(STR_TO_DATE(p.date_of_birth, '%m/%d/%Y')))) AS average_age
  FROM movie_people mp
         JOIN movie_data m ON m.imdb_title_id = mp.imdb_title_id
         JOIN people p ON p.imdb_name_id = mp.imdb_name_id
  WHERE mp.category = 'actress'
  GROUP BY mp.imdb_name_id, p.name, p.photo_url, p.date_of_birth
)
SELECT mp.imdb_name_id, ma.name, mc.total_movies, ma.photo_url,
       COUNT(*) AS oscar_freq,
       ROUND(MAX(om.year_ceremony - YEAR(ma.date_of_birth))) AS max_oscar_age,
       ROUND(AVG(m.avg_vote),1) AS avg_rating,
       ma.max_age,
       ma.average_age
FROM movie_people mp
       JOIN oscar_movies om ON om.imdb_title_id = mp.imdb_title_id
       JOIN movie_counts mc ON mc.imdb_name_id = mp.imdb_name_id
       JOIN movie_ages ma ON ma.imdb_name_id = mp.imdb_name_id
       JOIN movie_data m ON m.imdb_title_id = mp.imdb_title_id
WHERE mp.category = 'actress'
GROUP BY mp.imdb_name_id, ma.name, ma.max_age,ma.average_age,mc.total_movies, ma.photo_url
ORDER BY AVG(om.year_ceremony - YEAR(ma.date_of_birth)) DESC
  LIMIT ${offset}, ${pageSize};
```