Department of Electrical and Computer Engineering
ECE 656: Database Systems

Movie Finder

Justin Ly

j36ly@uwaterloo.ca

20968490

Yakov Yuzhakov

yyuzhako@uwaterloo.ca

20870187

Project repository: https://git.uwaterloo.ca/yyuzhako/movie-finder-project

Abstract — Since the invention of cinema, approximately five hundred thousand titles now exist for the popular form of media with no signs of slowing down. With increased accessibility through popular streaming platforms finding something to watch has become increasingly difficult. In this project, a relational database design was implemented to create a Movie Finder application. An Entity Relationship (ER) model was developed for the application and translated into a relational schema. Data was obtained from three Kaggle datasets that originate from IMDB, Rotten Tomatoes, and various streaming platforms to build the database. Various design issues were encountered such as table normalization, relevant attribute selection, and primary key choices for various entities. Primary data integration issues were inconsistencies in movie titles, release years and data duplication among the datasets. Movie Finder allows users to search for movies based on selected characteristics, obtain review/movie details, and maintain a watched list with their own personal reviews and ratings. Future development considerations include adding additional attributes to the recommendation function, social functionality by allowing users to see other watched lists and utilizing user attributes to autogenerate recommendations through data mining techniques such as association discovery or clustering user demographics with favorite movies.

1 Introduction

Movies are a popular form of media that is ever-growing. With hundreds of thousands of movies available, it is hardly surprising we can spend hours browsing to find a movie. On average, people spend 45 hours a year searching for something to watch [1]. There are many attributes that define movies such as genre, running time, rating, leading actors/actresses, and many more. When searching for a movie, users generally have an idea based on a combination of these aspects. The purpose of this project is to develop a movie querying application, Movie Finder, which utilizes a command-line interface to search for movies based on selected characteristics. The application allows users to read reviews, check movie details, provide ratings, and review movies they have recently watched. A variety of data sources originating from IMDB [2], Rotten Tomatoes [3], various streaming platforms [4] will be used to populate the database. These datasets are user-generated from Kaggle. IMDB is one of the largest online movie data sources that forms the foundation of our database, providing a variety of characteristics to search from. Rotten Tomatoes provides the basis for detailed review data, and availability on streaming platforms will be used to augment the movie recommendation's function. MySQL (*version 8.0.27)* will be utilized for the database on the server-side, and Python (*version 3.7)* will be used to implement the main client functionality.

2 ER Model

Figure 1 illustrates the ER model for the Movie Finder application. It includes the relevant entities, relationships, and attributes in our database system. Cardinality and weak entity sets are also shown.

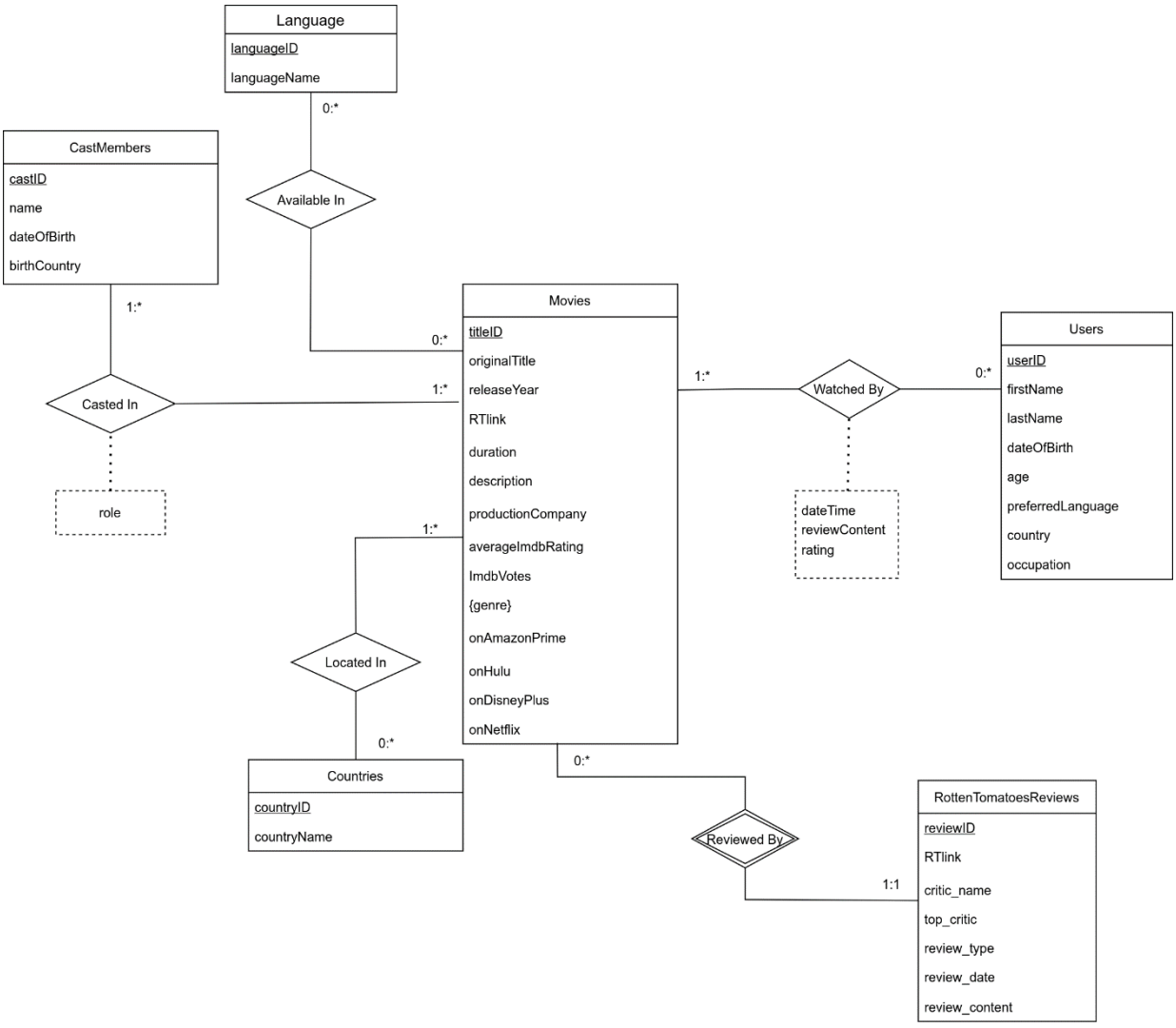

Fig.1 ER Diagram for the Movie Finder Application

3 Relational Schema

Based on the ER model shown above, a relational schema was created with normalized entities to integrate information about the movies, cast members, critics and reviews, users, watch history, countries, and languages. The final database consists of the following tables:

Tab.1 Tables in the project database

Table name	Description
Movies	The set of entities containing movie specific information such as title, release year, duration, description, movie production company, rating, genre and presence on different streaming platforms.
CastMembers	The set of entities describing people involved in movie production (includes names, countries of origin, dates of birth).
Countries	The set of different countries.
Language	The set of different languages.
RottenTomatoesReviews	The set of weak entities which includes information about reviews given to movies on the Rotten Tomatoes platform. Every review entity has review content, review type (fresh or rotten), date, critic name and a top critic indicator (represented in binary). A link to the movie on the Rotten Tomatoes platform is also included.
Users	The set of entities describing potential Movie Finder application users. Currently this includes a user's first and last names, preferred language, country of residence, date of birth and occupation. The set was randomly generated for the purpose of this project.
MovieUser	The relationship set representing user watch history. Consists of userID, titleID and descriptive attributes such as timestamp, user movie rating and user review. The set was randomly generated for the purpose of this project.
MovieCast	The relationship set representing the cast involved in the production of a certain movie. Consists of castID, titleID and a descriptive attribute – role in the movie (actor, producer, director, etc.).
MovieCountry	The relationship set representing the movie – country relation.
MovieLanguage	The relationship set representing the movie – language relation.

Test cases were written to ensure cross table data consistency and each table was also checked for appropriate attributes/data types. Database test cases include valid and invalid sample queries to retrieve information from each table, and data constraint tests (foreign keys, checks, triggers).

3.1 Movies table

To create the Movies table, values from 'IMDb movies.csv' were loaded with the following SQL command:

```
load data local infile './IMDb movies.csv' ignore into table Movies
```

…

'imdb_title_id' was chosen as a **primary key**, omitting the alphabetical characters in the original dataset (all the ids initially started with 'tt'). The **int** type is used for the primary key attribute instead of **char(9)**. This transformation was done to speed up queries for movies using the titleID.

As the **primary key** must be unique, specifying 'titleID' as a PK helped us to skip duplicate rows in the source file. At the time of loading all the empty title values were converted from empty string to NULL. Having specified the 'originalTitle' column as **not null,** empty entries into the database are prevented.

The 'originalTitle' attribute is set as **not null** to ensure each entry has a movie title specified. In addition, the Movies table has a 'processedTitle' attribute created based on the 'originalTitle' by lowercasing and removing non-alphanumeric characters. The main purpose of this attribute was to link different database sources (IMDb, Rotten Tomatoes, Netflix, Amazon, Disney, Hulu) together reducing possible typographical differences in the titles. This attribute is also used in the application, providing greater accuracy in search results from the user input. The final table has an index on 'processedTitle' to improve query operations on that attribute.

The initial Movies dataset had columns such as 'country' and 'language' with **char(40)** values from the IMDb source. These attributes were skipped after normalization to create MovieCountry and MovieLanguage relation tables. This was done to normalize the Movies table. Country and language attributes are also present in other entities; thus, normalization is beneficial for foreign key relations in other sets.

The 'genre' attribute is kept exclusively in movies, rather than in a separate normalized table as it can be represented with the **set** data type. The only limitation with **set**, is that it can only hold up to 64 distinct values. This fits well within the 25 possible genres defined in the IMDB dataset. Pre-defining the 25 possible genres at the stage of the table creation helps to reduce errors. The set attribute also allows multiple values to be defined for a single row, reducing repetition.

Streaming platforms attributes have a **tinyint** type to represent the presence of a movie on a certain online platform. These attributes were set to 0 until all streaming data is loaded. The attributes require a **binary** type, but this is not supported in MySQL, thus **tinyint** chosen as smallest integer type.

To add streaming data to the Movies table, a temporary table called 'tempStreamData' was created. The table was sequentially filled with titles and release years of movies available on every streaming platform - Amazon Prime, Hulu, Disney Plus and Netflix. All the titles were processed by lowercasing and removing non-alphanumeric characters. The temporary table was then inner joined with the Movies table using 'processedTitle' and the relevant movie attribute was updated from 0 to 1 indicating presence of the movie on the platform. After each update, the temporary table was emptied and fully dropped after all the data was loaded.

To link the movies table, to Rotten Tomatoes Reviews, a temporary table called RottenTomatoesMovies was created. The dataset *`rotten_tomatoes_movies.csv`* loaded into this table represents the comprehensive list of movies on the Rotten Tomatoes website with 'RTlink' as the unique movie identifier. Movie title and release year were also loaded for linking purposes. During loading, the title was processed to eliminate matching errors related to punctuation and spacing:

```
…
processedTitle = LOWER(REGEXP_REPLACE(@movie_title, '[^0-9a-zA-Z]', '')),
…
```

A temporary index was also added to the title column to reduce processing time for database linking queries. An issue identified at this stage was approximately 30% of movies in Rotten Tomatoes includes 2 titles in the title string (e.g. – '10 Nights(Ten Nights)'), while IMDb only has one original title. Therefore, linking two different movie databases was performed in three steps:

1. link movies with exact title/year matches (note: the attribute 'm.inRT' identifies if IMDb movie title has been linked with Rotten Tomatoes to skip those lines at the next steps):

```
UPDATE Movies m right join RottenTomatoesMovies rtm on m.processedTitle
=
    rtm.processedTitle
    SET rtm.titleID = m.titleID, m.inRT = 1
    WHERE m.releaseYear <= rtm.releaseYear + 1 and m.releaseYear >=
    rtm.releaseYear - 1;
```

2. link movies with multiple titles (note: regular expressions were not used here as some titles contain symbols not supported by REGEX):

```
with m_temp as (select * from Movies WHERE inRT is NULL),
    rt_temp as (select * from RottenTomatoesMovies where titleID is
NULL),
    temp as (select m.titleID, rtm.RTlink from m_temp m inner join
        rt_temp rtm
        ON (rtm.processedTitle LIKE Concat('%',m.processedTitle) or
        rtm.processedTitle LIKE Concat(m.processedTitle,'%'))
        WHERE m.releaseYear <= rtm.releaseYear + 1 and m.releaseYear >=
        rtm.releaseYear - 1)
UPDATE RottenTomatoesMovies r inner join temp t using (RTlink)
    SET r.titleID = t.titleID;
```

3. link movies with missing release year in one of the source databases (approx. 3% of all movies):

```
UPDATE Movies m right join RottenTomatoesMovies rtm on m.processedTitle
= rtm.processedTitle
    SET rtm.titleID = m.titleID, m.inRT = 1
    WHERE rtm.titleID is NULL;
```

The described method above allows >90% of all movies in IMDb, Rotten Tomatoes and streaming platform datasets to be linked. The RottenTomatoesMovies table is subsequently dropped after linking.

The remaining attributes have appropriate data types chosen based on data observation.

The Movies table also includes checks to ensure movie duration (mins) is greater than 0, and rating is specified within a valid range (0-10).

3.2 CastMembers table

This table loads data from the 'IMDb names.csv' file. Similar to Movies, the numeric portion of 'imdb_name_id' was chosen as a **primary key** to identify each cast member.

The primary issue with CastMembers was extracting birth country from the source. From data observation, in approximately 90% of the rows, country is the last word in 'place of birth' with or without parenthesis. Thus, the following code was used to extract birth country data:

```
birthCountry = NULLIF(REGEXP_REPLACE(SUBSTRING_INDEX(@place_of_birth, ' ', -1), '[^A-Za-z ]', ''), '')
```

…

This removes punctuation characters and extracts the last word separated by a space character. A **foreign key** constraint on birthCountry referencing the Countries table was added to reduce data entry errors in this column. An index was added to the name attribute for expected client queries related to cast member search.

3.3 Countries and Language tables

The main source for these two tables were the 'country' and 'language' columns of 'IMDb movies.csv'. All distinct entities separated by a comma, or a new line are assigned unique integer IDs. The resultant tables have 265 distinct countries and 267 distinct languages, which serve as referenced **foreign keys** for other tables such as CastMembers or Users. These foreign keys reduce typographical errors for language or country names when adding new users, cast members, or movies. An index was placed on countryName and languageName to expedite expected queries in the client.

3.4 RottenTomatoesReviews table

This table is primarily populated by `rotten_tomatoes_critic_reviews.csv' and is a weak entity dependent on Movies. The primary key, along with the original data attributes as discriminators could not ensure uniqueness in review entries. Although it is unlikely when using all original attributes as discriminators, it is still possible for a row to contain duplicate information. Therefore, the attribute 'reviewID' was added with an auto increment parameter. Additionally, this makes it easier to reference the given review in future application development (searching for a particular review, for example).

```
create table RottenTomatoesReviews (
    reviewID int primary key AUTO_INCREMENT,
    RTlink char(80) NOT NULL,
    …
```

The 'RTlink' attribute was set as **not null** to skip empty rows in source data and ensure future database consistency as this attribute is the main movie identifier in Rotten Tomatoes database. This attribute is set as a **foreign key** referencing Movies.

'top_critic' and 'review_type' were set as **tinyint** due to MySQL limitations, ideally those should be **binary** to tell us if it is a top critic review or not, and if a reviewer concludes the movie is 'fresh' or 'rotten' overall.

To prevent the addition of a review written earlier than the movie's release year (case for movies with similar titles) we included a cross table check on dates using a 'before insert' trigger in MySQL:

```
CREATE TRIGGER dateReview BEFORE INSERT ON RottenTomatoesReviews
FOR EACH ROW
    IF YEAR(new.review_date) < (SELECT m.releaseYear from Movies m
        inner join RottenTomatoesMovies rtm using(titleID)
        where rtm.RTlink = new.RTlink)
    THEN SIGNAL SQLSTATE '50001' SET MESSAGE_TEXT = 'Review must be written
after movie release year.';
    END IF; //
```

3.5 Users table

To develop and test the application, the Users table was filled with random data of 400 names, dates, countries, etc. The 'userID' attribute was created with the **int** type and autoincrement parameter and acts as the **primary key**. 'age' is calculated based on date of birth. Three columns, dateOfBirth, age and country, were set to **not null** and involved in **check** constraints:

```
    CHECK (age > 17),
    CHECK (country <> 'North Korea' or 'Syria')
```

The first check was established due to adult titles being present in the database. The second check was established as local regulations do not allow the usage of a product in certain countries. [5]

Foreign key constraints include 'country' referencing the Countries table, and 'preferredLanguage' referencing the Language table.

The rest of the attributes have appropriate data types based on data observation.

3.6 MovieUser table

MovieUser also contains randomly generated data representing the watch history of all users in the database. Each entry is uniquely identified by titleID, userID as the **primary key**. Users can optionally provide a rating and review for watched movie.

3.7 MovieCast table

This table was populated using `title_principals.csv` and identifies who was involved in the production of a movie along with their primary role. The **primary keys** are titleID, castID and role is an additional attribute for the relation. Both attributes titleID and castID also act as **foreign keys** referencing Movies and CastMembers, respectively.

3.8 MovieCountry and MovieLanguage tables

These tables represent normalized relations between movies and countries/languages. They were populated with the following code:

```
INSERT INTO MovieCountry (titleID, countryID)
    select m.titleID, c.countryID from Movies m join Countries c
    where m.country like CONCAT('%', c.countryName, '%');


INSERT INTO MovieLanguage (titleID, languageID)
    select m.titleID, l.languageID from Movies m join Language l
    where m.language like CONCAT('%', l.languageName, '%');
```

This relational schema allows us to link multiple countries/languages with multiple movies and vice versa without repeating information within Movies table. Each relation is represented just by two integers thus saving disk space.

4 Client Application

4.1 Ideal Client

The ideal client application consists of three core functionalities:

1. find movie recommendations, based on a combination of selected characteristics;
2. find additional details about a movie through reviews or a detailed description; and
3. allow a user to maintain a watched list, containing their own rating/review data.

Ideally, the application would be a web-based graphical interface for ease-of-use and user friendliness. Due to time limitations, a minimum viable product was created based on a command-line interface. User inputs should also be validated with regular expressions, however for this project only basic validation and input sanitization are implemented, assuming non-malicious, valid inputs. User registration and authentication procedures should also be present, but due to time constraints users will be autogenerated and authentication will be omitted.

4.2 Proposed Client

The main user interface will consist of menu options that present the core functionalities. With the first core functionality, a user is presented with a variety of characteristics to search from. These choices present prompts that formulate subqueries related to that characteristic. When the user decides to search, the query will execute finding all matching titles related to the combination of movie characteristics. The second functionality emulates selection queries based on a given movie name. The final functionality allows users to update, delete and insert values in the database through a watched list.

4.3 Implementation

Figure 2 shows the main menu options available to the user upon execution of the main python file (*main.py)*. The client implementation has functionality for all core concepts noted above. Query implementation details for the client can be found in Appendix A. Option 1 encapsulates the first core functionality, Option 2, and 4 demonstrates the second functionality, and Option 3, 5, 6 captures the last functionality.

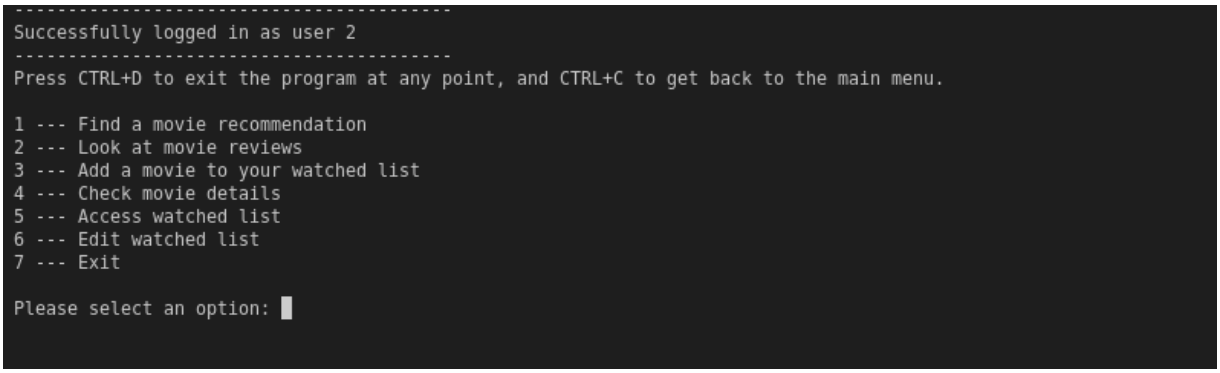

Fig.2 Command line interface menu on start-up of the application

Upon selection of the first option, users can input various parameters to formulate subqueries. At least one input must be added before executing a search. Users can find movies based on the following:

- a list of cast members who were in the same film;
- a list of directors who worked on the same film;
- movies before/after/equal a certain year;
- movies shorter/longer than a specified duration;
- movies greater/less than a specified IMDB rating;
- a list of countries the movie was in;
- a list of languages the movie has;
- a list of genres that describe the movie;
- availability on streaming platforms; and
- movies with at least a 60% fresh rating on Rotten Tomatoes by at least five top critics.

Upon search, each subquery is inner joined with the Movies table and ten results are displayed for readability with the option to fetch more results if they exist. Users can query a movie by title using the second and fourth option to access the relevant reviews and details from Rotten Tomatoes and IMDB data, respectively. Users also have access to a watched list where they can view, add, edit, and delete movies on their watched list by querying a movie by title. Movies on the watched list contain the users' personal rating out of ten and their review of a movie. Query performance is nearly instantaneous for all three functionalities due to effective database design, and appropriate indexes put in place in the schema.

Unit test cases were developed to verify the client functionality. The test cases run through a variety of sample queries to ensure adequate coverage and reliability of the client. The test cases are considered successful if the functional procedures result in a successful exit operation with the correct output.

4.4 Future Development

Through further development, features can be added that utilize additional attributes in the database. A few examples are as follows.

- find a movie based on cast members' country of birth;
- incorporation of other roles (writer, cinematographer, producer) into the query process;
- search for a movie based on production company; and
- search for movies watched by a certain critic;

User watched lists could also be made accessible to other users, adding a social feature to the application. These features were omitted in the implementation as core concepts were sufficiently illustrated in the current program. User attributes could also be used for generating recommendations through data mining techniques such as association discovery or clustering user demographics with movies they liked.

5 Conclusion

In this project, concepts of database design and implementation were explored by creating a movie querying application (Movie Finder). An ER model was created and translated into a relational schema for the application. Datasets from IMDB, Rotten Tomatoes, and various streaming platforms were used to populate the database. Various design issues such as choice of primary key, table normalization, and attribute selection were encountered in the database design. Primary issues with integrating data sources were title/year inconsistencies, and data duplication. The command-line application allows users to search for movies based on selected characteristics. In addition, Movie Finder can read detailed reviews, obtain additional movie details, and allow users to maintain a watched list with their own ratings/reviews. All proposed core functionalities were implemented in the application. Further development opportunities include incorporating additional attributes into the search process and utilizing user attributes for data mining to auto-generate recommendations.

References

[1] https://www.vox.com/ad/20974139/streaming-content-movies-tv-shows-algorithm-human-choice

[2] https://www.kaggle.com/stefanoleone992/imdb-extensive-dataset

[3] https://www.kaggle.com/stefanoleone992/rotten-tomatoes-movies-and-critic-reviews-dataset

[4] https://www.kaggle.com/ruchi798/movies-on-netflix-prime-video-hulu-and-disney

[5] https://www.international.gc.ca/world-monde/international_relations-relations_internationales/sanctions/types.aspx

Appendix A: SQL queries related to client functionality

Values surrounded by {} are user inputs:

Core Functionality 1: Movie Recommendations

1.1 Search by Cast Members:

SELECT titleID FROM MovieCast

INNER JOIN CastMembers USING(castID)

INNER JOIN Movies USING(titleID)

WHERE role IN ('actress','actor')

AND name IN ({Actor1,Actor2})

1.2 Search by Directors:

SELECT titleID From MovieCast

INNER JOIN CastMembers USING(castID)

INNER JOIN Movies USING(titleID)

WHERE role IN ('director')

AND name IN ({director1,director2})

1.3 Search by Year:

SELECT titleID FROM Movies WHERE releaseYear {>=,<=,=} {year}

1.4 Search by Duration:

SELECT titleID FROM Movies WHERE duration {>=,<=,=} {duration}

1.5 Search by Rating:

SELECT titleID FROM Movies WHERE averageImdbRating {>=,<=,=} {ImdbRating}

1.6 Search by Country:

SELECT titleID FROM MovieCountry

INNER JOIN Countries USING (countryID)

WHERE countryName IN ({country1,country2})

1.7 Search by Language:

SELECT titleID From MovieLanguage

INNER JOIN Language USING (languageID)

WHERE languageName IN ({language1,language2})

1.8 Search by Genre:

SELECT titleID From Movies WHERE find_in_set({genre1},genre) AND find_in_set({genre2},genre) AND...

1.9 Search by RT Fresh:

SELECT RTLink, titleID, (sum(review_type = 1) / (sum(review_type = 1) + sum(review_type = 0))) as percentFresh

FROM RottenTomatoesReviews INNER JOIN RottenTomatoesMovies USING (RTLink)

WHERE top_critic = 1

GROUP BY RTLINK HAVING percentFresh >= 0.6 AND count(top_critic) > 5

1.10 Search by Streaming Platform:

SELECT titleID FROM Movies WHERE onAmazonPrime = {0,1}

SELECT titleID FROM Movies WHERE onHulu = {0,1}

SELECT titleID FROM Movies WHERE onNetflix = {0,1}

SELECT titleID FROM Movies WHERE onDisneyPlus = {0,1}

1.11 Search Combination:

***** Each option listed above is optional in the combination SQL query

WITH A as (1.1 query), B as (1.2 query), C as (1.3 query), D as (1.4 query), E as (1.5 query),

F as (1.6 query), G as (1.7 query), H as (1.8 query), I as (1.9 query), J as (1.10 query)

SELECT originalTitle, releaseYear, genre, duration,

averageImdbRating, ImdbVotes, onHulu, onDisneyPlus,

onAmazonPrime, onNetflix FROM Movies INNER JOIN

A USING (titleID) INNER JOIN B USING (titleID) INNER JOIN C USING (titleID)

INNER JOIN D USING (titleID) INNER JOIN E USING (titleID) INNER JOIN F USING (titleID)

INNER JOIN G USING (titleID) INNER JOIN H USING (titleID) INNER JOIN I USING (titleID)

INNER JOIN J USING (titleID)

LIMIT 10 OFFSET {offset};

<u>Core Functionality 2: Additional Movie Details</u>

2.0 Movie Title Confirmation:

SELECT titleID, originalTitle, releaseYear FROM Movies WHERE processedTitle LIKE {titleInput};

2.1 Get Movie Reviews:

SELECT critic_name, top_critic, review_type, review_date, review_content

FROM RottenTomatoesReviews

WHERE RTlink = (SELECT RTlink FROM RottenTomatoesMovies WHERE titleID='{matching_row[0]}')

LIMIT 10 OFFSET {offset};

2.2 Get Movie Details:

SELECT originalTitle, releaseYear, genre, duration, averageImdbRating,

ImdbVotes, onNetflix, onHulu, onDisneyPlus, onAmazonPrime, productionCompany, description

FROM Movies where titleID = {titleID}

Core Functionality 3: Watched List

3.0 Movie Title Confirmation:

SELECT titleID, originalTitle, releaseYear FROM Movies WHERE processedTitle LIKE {titleInput};

3.1 Add to Watched List:

INSERT INTO MovieUser (titleID, userID, rating, reviewContent) VALUES ({titleID},{userID},{rating},{reviewContent});'

3.2 Get Watched List:

SELECT originalTitle, releaseYear, duration, rating, reviewContent

FROM MovieUser

INNER JOIN Movies USING (titleID) WHERE MovieUser.userID = {userID} ORDER BY dateTime ASC;

3.3 Edit Watched List Item:

UPDATE MovieUser SET rating = {rating}, reviewContent = {reviewContent}

WHERE titleID = {titleID} and userID = {userID};

3.4 Delete Watched List Item:

DELETE FROM MovieUser

WHERE titleID = {titleID} AND userID = {userID};