Introduction and Dataset Overview

The dataset used for this project was sourced from Kaggle datasets and contains information about movies available on Netflix, a renowned media and video streaming platform. This tabular dataset provides comprehensive listings of movies, including essential details such as cast members, directors, ratings, release years, durations, genres, production countries, date added to Netflix, and descriptions for each film.

Our goal is to provide essential information about the films, facilitating easy searches based on various criteria such as release year, duration, actors, directors, production countries, and genres. Users will have the option to search for movies released in specific years, filter by preferred duration, explore films featuring specific actors in particular genres, discover works by favorite directors, or explore movies produced in specific countries.

Database Schema Structure:

MOVIES Table Documentation

Overview

The MOVIES table stores information about movies, including details such as title, release year, rating, duration, and description.

Table Structure

The MOVIES table has the following structure:

Column Name	Data Type	Description
\`movie_id\`	int(11)	Unique identifier for each movie..
\`director_name\`	varchar(100)	Title of the movie.
\`release_year\`	int(11)	Year when the movie was released.
\`rating\`	varchar(100)	Rating of the movie.
\`duration\`	int(11)	Duration of the movie in minutes.
\`description\`	varchar(1500	Description or summary of the movie

Constraints

- **Primary Key**: The movie_id column serves as the primary key, ensuring each movie has a unique identifier.
- **Indexes**: Additional indexes have been created to enhance query performance. B-tree indexes have been chosen for their effectiveness in handling queries that involve ranges, such as numerical ranges for duration and release year.
- **Full-text Index**: A full-text index named idx_description is created on the title and description columns to enable full-text searches on movie titles and descriptions.

Modification in Database Design:
In the "MOVIES" table, the "date_added" column has been removed from the schema. This decision was made as the "date_added" information was not utilized in any of the queries or the application functionality. By removing this column, we have streamlined the database schema, reducing redundancy and optimizing database performance.

DIRECTORS Table Documentation

Overview

The DIRECTORS table establishes a many-to-many relationship between movies and directors. It stores information about which directors are associated with each movie.

Table Structure

The DIRECTORS table has the following structure:

Column Name	**Data Type**	**Description**
`movie_id`	int(11)	Foreign key referencing the `movie_id` in the `MOVIES` table.
`director_name`	varchar(100)	Stores the name of the director associated with the movie.

Constraints

- **Primary Key:** The combination of director_name and movie_id serves as the composite primary key for the table, ensuring uniqueness of combinations of director and movie.
- **Foreign Key:** Establishes a foreign key constraint on movie_id, referencing the movie_id column in the MOVIES table.

Indexes
No additional indexes are explicitly defined in the `DIRECTORS` table. However, the primary key index implicitly ensures fast lookup and uniqueness enforcement for the primary key constraint.

CAST Table Documentation

Overview

The CAST table stores information about the cast members associated with each movie. It establishes a relationship between movies and their cast members.

Table Structure

The CAST table has the following structure:

Column Name	Data Type	Description
\`movie_id\`	int(11)	Foreign key referencing the \`movie_id\` in the \`MOVIES\` table.
\`cast_name\`	varchar(100)	Stores the name of the cast member.

Constraints

- **Primary Key:** The combination of cast_name and movie_id serves as the composite primary key for the table, ensuring uniqueness of combinations of cast member and movie.
- **Foreign Key:** Establishes a foreign key constraint on movie_id, referencing the movie_id column in the MOVIES table.

Indexes

- A FULLTEXT index named idx_cast_name is created on the cast_name column, enabling efficient full-text searches on cast member names.

COUNTRY Table Documentation

Overview

The COUNTRY table stores information about the countries associated with each movie. It establishes a relationship between movies and their respective countries.

Table Structure

The COUNTRY table has the following structure:

Column Name	Data Type	Description
`movie_id`	int(11)	Foreign key referencing the `movie_id` in the `MOVIES` table.
`country_name`	varchar(100)	Stores the name of the country.

Constraints

- **Primary Key:** The combination of country_name and movie_id serves as the composite primary key for the table, ensuring uniqueness of combinations of country and movie.
- **Foreign Key:** Establishes a foreign key constraint on movie_id, referencing the movie_id column in the MOVIES table.

Indexes

No additional indexes are explicitly defined in the `COUNTRY ` table. However, the primary key index implicitly ensures fast lookup and uniqueness enforcement for the primary key constraint.

LISTED_IN Table Documentation

Overview
The `LISTED_IN` table stores information about the categories or genres in which movies are listed. It establishes a relationship between movies and the categories or genres they belong to.

Table Structure
The `LISTED_IN` table has the following structure:

Column Name	Data Type	Description
`movie_id`	int(11)	Foreign key referencing the `movie_id` in the `MOVIES` table.
`listed_in_name`	varchar(100)	Stores the name of the category or genre in which the movie is listed.

Constraints
- **Primary Key:** The combination of `movie_id` and `listed_in_name` serves as the composite primary key for the table, ensuring uniqueness of combinations of movie and category.
- **Foreign Key:** Establishes a foreign key constraint on `movie_id`, referencing the `movie_id` column in the `MOVIES` table.

Indexes
No additional indexes are explicitly defined in the `LISTED_IN` table. However, the primary key index implicitly ensures fast lookup and uniqueness enforcement for the primary key constraint.

****

Database Schema Overview:

It was crucial to separate the table into more tables, particularly to create a dedicated table for cast. This decision was vital because each film features a long list of actors. When searching for a specific actor, we would need to check all the listings for every movie in the original table, which is not efficient. Therefore, splitting the table for the cast into separate entities became imperative.
We encountered a similar situation with directors, prompting us to also split them into a new table.
For production countries, there is also a list for every film. Splitting this information is important to ensure efficient searches.
Lastly, for searching specific genres, we had to split a new table for easier searches for the same reasons.
However, we believe it's acceptable to leave the remaining film details in the same table as the movies, given its good and efficient search functionality

The database schema also includes several tables designed to efficiently store and manage movie-related data. In the MOVIES table, the primary key is established on the movie_id column to ensure each movie has a unique identifier. Additionally, three indexes have been added to enhance query performance: idx_description is a full-text index covering the title and description columns, enabling efficient keyword searches within movie titles and descriptions. Moreover, indexes have been created on the duration and release_year columns, named duration_index and release_year_index respectively, facilitating faster retrieval of movies based on their duration and release year. The DIRECTORS, CAST, COUNTRY, and LISTED_IN tables also utilize composite primary keys to maintain data integrity, with each including a combination of the movie_id column and another relevant column (director_name, cast_name, country_name, and listed_in_name respectively). These primary keys implicitly create indexes on the associated columns to optimize query performance and enforce uniqueness constraints.

Notes on Indexing in MySQL

- ❖ MySQL requires that foreign key columns be indexed. If we create a table with a foreign key constraint but no index on a given column, MySQL automatically creates an index on that column. This indexing ensures efficient querying and maintenance of referential integrity in relational databases.
- ❖ When a composite primary key is defined on columns director\cast\country\listed_in_name and movie_id, MySQL automatically creates an index on the director\cast\country\listed_in_name column as part of enforcing the primary key constraint. Therefore, an index exists on the director\cast\country\listed_in_name column even though it may not be explicitly defined.

Relationships

The DIRECTORS, CAST, COUNTRY, LISTED_IN tables represent a many-to-many relationship between movies and director\cast\country\listed_in.

Note: CSV File Overview

The CSV file consists of 8000 rows and 9 columns, encompassing both TV shows and movies from Netflix. To streamline our analysis, we filtered the dataset, extracting only movie entries for further examination. This focused approach allows us to delve specifically into movie-related data, facilitating more targeted insights and analysis.

The dataset provided, originally comprising both TV shows and movies available on Netflix, underwent a meticulous filtering process to exclusively isolate movie entries. This filtering operation was executed within the Python script

Detailed Analysis of Main Queries

The 5 main queries:

Query 1: Search Movies by Title and Description

```
SELECT title, release_year, rating, duration, description
FROM MOVIES
WHERE MATCH(title, description) AGAINST (%s IN NATURAL LANGUAGE MODE)
ORDER BY release_year DESC, title ASC
```

- **Purpose:** Retrieve movie details based on a keyword search in the title or description. Additionally, the query aims to order the movies by their release year in descending order and then by title in ascending order.

- **Optimization Strategy:**Utilizes full-text search capability for efficient keyword search on the title and description columns, leveraging multiple indexes.

Query 2: Search Movies by Cast Name

```
SELECT CAST.cast_name, MOVIES.title, MOVIES.release_year, MOVIES.description
FROM MOVIES
JOIN CAST ON MOVIES.movie_id = CAST.movie_id
WHERE MATCH(cast_name) AGAINST (%s IN NATURAL LANGUAGE MODE)
ORDER BY CAST.cast_name ASC, release_year DESC
```

- **Purpose:** Retrieve movies along with their details where a specific cast member is involved. Additionally, the query aims to order the results alphabetically by the cast member's name in ascending order and by release year in descending order.
- **Optimization Strategy:** Utilizes full-text search capability for efficient retrieval of movies based on cast members' names, leveraging the index on the cast_name column.

Query 3: Retrieving Movies Directed by the Top Director

```
SELECT d.director_name, m.title, m.release_year, m.rating, m.duration, m.description
FROM DIRECTORS d
JOIN (
   SELECT d.director_name, COUNT(m.movie_id) AS num_movies
   FROM DIRECTORS d
   JOIN MOVIES m ON d.movie_id = m.movie_id
   GROUP BY d.director_name
   ORDER BY num_movies DESC
   LIMIT 1
) best_director ON best_director.director_name = d.director_name
JOIN MOVIES m ON d.movie_id = m.movie_id
ORDER BY release_year DESC
```

- **Purpose:** The purpose of this query is to retrieve the movies directed by the director who has directed the maximum number of movies in the dataset. Additionally, the query aims to order the movies by their release year in descending order.
- **Optimization Strategy**: The primary key index on director_name in the DIRECTORS table ensures fast retrieval of director information during joins, while indexing movie_id facilitates data integrity and enhances query performance. Additionally, applying ORDER BY and LIMIT clauses in the subquery limits the result set to the director with the maximum number of movies, optimizing query execution by focusing on relevant data.

Query 4: Retrieve Movies with Shortest Duration in a Specific Category

```
SELECT MOVIES.title, MOVIES.duration, MOVIES.description
FROM MOVIES as m JOIN LISTED_IN as li on m.movie_id = li.movie_id
WHERE LISTED_IN.listed_in_name = %s
AND MOVIES.duration <= ALL (
   SELECT MOVIES.duration
   FROM MOVIES as m JOIN LISTED_IN as li on m.movie_id = li.movie_id
   AND LISTED_IN.listed_in_name = %s)
```

- **Purpose:** Retrieve the shortest movie in a specific genre.
- **Optimization Strategy:** Utilizes JOIN operation between the MOVIES and LISTED_IN tables for efficient retrieval of movie details and genre information. Filters the result set to include movies belonging to the chosen genre, ensuring efficient utilization of indexes. Enhances query performance by utilizing a subquery to find the shortest duration movie within the chosen genre, efficiently leveraging the index on the duration column.

Query 5: Calculate Genre Percentage by Country

```
SELECT li.listed_in_name AS genre,
        ROUND((COUNT(li.listed_in_name) / total_movies.total_count) * 100, 2) AS percentage
   FROM COUNTRY c
   JOIN LISTED_IN li ON c.movie_id = li.movie_id
   JOIN (
      SELECT country_name, COUNT(movie_id) AS total_count
      FROM COUNTRY
      GROUP BY country_name
   ) total_movies ON c.country_name = total_movies.country_name
   WHERE c.country_name = %s
   GROUP BY  li.listed_in_name
   ORDER BY percentage DESC
```

- **Purpose:**This query calculates the percentage of each movie genre in a specific country. It retrieves the genre name and its corresponding percentage representation based on the number of movies in each genre relative to the total number of movies in the country.

- **Optimization Strategy:** The database design includes indexes on both the "country_name" column in the "COUNTRY" table and the "movie_id" column, which serves as a foreign key referencing the "movie_id" column in the "MOVIES" table. These indexes optimize the query's performance by facilitating efficient data retrieval and join operations. dditionally, the use of aggregation functions and grouping streamlines the calculation of genre percentages, further optimizing query performance.

Database Design Support for the Queries 2-5: The database design supports this query by establishing foreign key constraints between the movie_id columns of both tables. These constraints ensure the integrity of the relationship between movies and their listed categories.

Additional Queries:

Query 6: Retrieve Movie Information

```
SELECT title, release_year, rating, duration, description
FROM MOVIES
ORDER BY release_year DESC, title ASC
```

Purpose: This query allows users to access detailed information about every movie available on the Netflix platform. Additionally, the query aims to order the movies by their release year in descending order and then by title in ascending order.

Query 7: Retrieve Short Movies

```
SELECT title, release_year, rating, duration, description
FROM MOVIES
WHERE duration < %s
ORDER BY duration DESC
```

Purpose: This query retrieves specific information about movies stored in the database with a duration shorter than a specified threshold. It selects the title, release year, rating, duration, and description of each qualifying movie from the "MOVIES" table. Additionally, the query aims to order the results by duration in descending order
Optimization Strategy: The duration column in the MOVIES table is indexed.

Query 8: Retrieve Movies by Rating

```
SELECT title, release_year, duration,  description
FROM MOVIES
WHERE rating = %s
```

Purpose: This query retrieves specific information about movies stored in the database that match a particular rating. It selects the title, release year, duration, and description of each movie from the "MOVIES" table that corresponds to the specified rating.

Query 9: Retrieve Movies by Release Year Range

```
SELECT title, release_year, rating, duration, description
FROM MOVIES
WHERE release_year BETWEEN %s AND %s
```

Purpose: This query retrieves specific information about movies stored in the database that were released within a specified range of years. It selects the title, release year, rating, duration, and description of each movie from the "MOVIES" table that falls within the specified release year range.
Optimization Strategy: The release_year column in the MOVIES table is indexed.

Query 10: Retrieve Movies Directed by a Specific Director

```
SELECT m.title, m.release_year, m.rating, m.duration, m.description
FROM MOVIES m
JOIN DIRECTORS d ON m.movie_id = d.movie_id
WHERE d.director_name = %s
```

Purpose: This query retrieves specific information about movies stored in the database that were directed by a particular director. It selects the title, release year, rating, duration, and description of each movie from the "MOVIES" table.
Optimization Strategy: The director_name column in the DIRECTORS table is indexed by the primary key.

Query 11: Retrieve Movies Directed by a Specific Country

```
SELECT m.title, m.release_year, m.rating, m.duration, m.description
FROM MOVIES m
JOIN COUNTRY c ON m.movie_id = c.movie_id
WHERE c.country_name = %s
```

Purpose: This query retrieves specific information about movies stored in the database that belong to a particular country. It selects the title, release year, rating, duration, and description of each movie from the "MOVIES" table.
Optimization Strategy: The country_name column in the COUNTRY table is indexed by the primary key.

Query 12: Retrieve Movies Listed Under a Specific Category

```
SELECT m.title, m.release_year, m.rating, m.duration, m.description
FROM MOVIES m
JOIN LISTED_IN l ON m.movie_id = l.movie_id
WHERE l.listed_in_name = %s
```

Purpose: This query retrieves specific information about movies stored in the database that are listed under a particular category or genre. It selects the title, release year, rating, duration, and description of each movie from the "MOVIES" table.
Optimization Strategy: The listed_in_name column in the LISTED_IN table is indexed by the primary key.

The name and Link for the dataset:

Netflix Movies and TV Shows
Listings of movies and tv shows on Netflix - Regularly Updated
https://www.kaggle.com/datasets/shivamb/netflix-shows

Code Structure:

Database Setup:

- create_db_script.py: Contains SQL statements to create database tables for movies, directors, cast, country, and listed categories.

Data Retrieval and Preparation:

- api_data_retrieve.py: Handles retrieval and preparation of movie data from an external source (CSV file). Parses the data and organizes it into lists for populating the database tables.

Database Interaction:

- queries_db_script.py: Defines SQL queries for various operations on the database, such as retrieving movie details based on full-text search, fetching movies by cast, counting movies directed by each director, etc.
- queries_execution.py: Executes database-related tasks, including creating tables, inserting data, and executing queries. Provides examples of how to execute each query function defined in queries_db_script.py.

API Usage:

The API provides endpoints for common operations related to movies, such as searching for movies by title, retrieving movie details, counting movies directed by a director, finding the shortest movie in a genre, and calculating the average duration of movies by country and year. Clients can interact with these endpoints using HTTP requests to access and manipulate movie data in the application

API Usage Documentation

Introduction:
This document provides an overview of the API functions available for interacting with the movie database.

Function Overview:
- example_1(movie_search)
- example_2(cast_name)
- example_3()
- example_4(genre)
- example_5(country_name)
- example_6()
- example_7(year)
- example_8(rating)
- example_9(start_year, end_year)
- example_10(director_name)
- example_11(country)
- example_12(genre)

Function Details:

example_1(movie_search)
- **Purpose:** Retrieve movie details by providing a keyword or phrase.
- **User Interaction:** Users provide a keyword or phrase (movie_search) to search for in movie titles or descriptions.
- **Returns:** A list of movie details matching the provided keyword or phrase, ordered by release year in descending order and title in ascending order.

example_2(cast_name)
- **Purpose:** Retrieve movie details by providing a cast member's name.
- **User Interaction:** Users specify the name of a cast member (cast_name) they are interested in.
- **Returns:** A list of movies featuring the specified cast member, ordered by cast name in ascending order and release year in descending order.

example_3()

- **Purpose:** Retrieve all movies directed by the director with the most movies directed (top director).
- **User Interaction:** Users invoke this function without providing any parameters.
- **Returns:** A list of lists containing movie details directed by the top director, ordered by release year in descending order.

example_4(genre)

- **Purpose**: Retrieve the shortest movie in a specified genre.
- **User Interaction:**Users select a genre for which they want to find the shortest movie.
- **Returns**: Details of the shortest movie in the specified genre.

example_5(country_name)

- **Purpose:** Retrieve the percentage representation of each movie genre in a specific country.
- **User Interaction:** Users provide the country_name parameter specifying the name of the country.
- **Returns:** A list of lists containing each movie genre and its corresponding percentage representation in the specified country.

example_6()

- **Purpose:** Retrieve detailed information about all movies available on the Netflix platform.
- **User Interaction:** Users invoke this function without providing any parameters.
- **Returns:** A list of lists containing detailed information about each movie, ordered by release year in descending order and title in ascending order.

example_7(year)

- **Purpose:** Retrieve specific information about movies stored in the database that have a duration shorter than a specified threshold.
- **User Interaction:** Users provide the year parameter specifying the maximum duration allowed for the movies to be retrieved.
- **Returns:** A list of lists containing movie details meeting the specified duration threshold.

example_8(rating)

- **Purpose:** Retrieve specific information about movies stored in the database that match a particular rating.
- **User Interaction:** Users provide the rating parameter specifying the rating of the movies to retrieve.
- **Returns:** A list of lists containing movie details matching the specified rating.

example_9(start_year, end_year)

- **Purpose:** Retrieve specific information about movies stored in the database that were released within a specified range of years.
- **User Interaction:** Users provide the start_year and end_year parameters specifying the range of years.
- **Returns:** A list of lists containing movie details released within the specified range of years.

example_10(director_name)

- **Purpose**: Retrieve specific information about movies stored in the database that were directed by a particular director.
- **User Interaction**: Users provide the director_name parameter specifying the name of the director.
- **Returns:** A list of lists containing movie details directed by the specified director.

example_11(country)

- **Purpose:** Retrieve specific information about movies stored in the database that belong to a particular country.
- **User Interaction**: Users provide the country parameter specifying the name of the country.
- **Returns:** A list of lists containing movie details produced in the specified country.

example_12(genre)

- **Purpose**: Retrieve specific information about movies stored in the database that are listed under a particular genre.
- **User Interaction:** Users provide the genre parameter specifying the name of the genre.
- **Returns**: A list of lists containing movie details categorized under the specified genre.

General Flow of the Application:

User Interaction:

- User interacts with the frontend interface (not provided), such as searching for movies, filtering by genre, etc.

HTTP Requests:

- Frontend makes HTTP requests to the backend API endpoints based on user actions, passing necessary parameters.

Server-side Processing:

- Backend receives HTTP requests and routes them to appropriate handlers/controllers.

Database Interaction:

- Handlers/controllers execute the corresponding database queries using functions defined in queries_db_script.py.
- Query parameters are passed to the query functions, and the results are retrieved from the database.

Response Generation:

- Backend generates HTTP responses containing query results.
- Responses are sent back to the frontend for display to the user.

Display to User:

- Frontend receives HTTP responses and updates the UI accordingly, displaying movie information, search results, etc.