בס״ד

DBMS – Assignment 3
System Documentation

Milana Yakubov, 213400369
Guy Harem, 312576655

Overview:

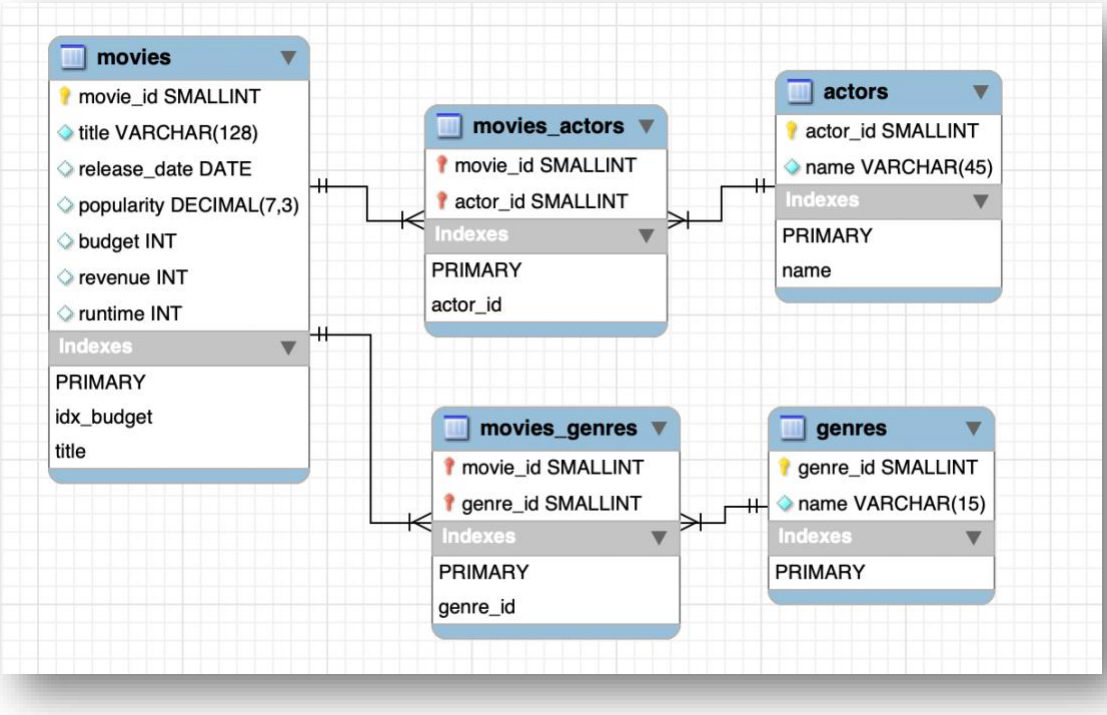
In this assignment we were asked to implement a web application that is centered on movies. We chose to tailor our program to the movie producer's community, trying to answer their needs on maximizing their movies profits.

We decided to focus our program on a few key features that usually results in the producers the best revenue for their movie (e.g. movie's budget, revenue, runtime, popularity, cast).

Our Database relies only on data from API requests from TMDB (https://developer.themoviedb.org/reference/intro/getting-started), ensuring the most up to date movie information, with minimal data errors as possible. The implementation does not support manually entered data by the user, and for such does not support any errors that can result from this.

Each of our queries is meant to guide the user (movie producer) to make a better decision on making their movie the most profitable they can, each assist in covering a different aspect of the movie (cast, budget, genre, runtime and release date).

Database Schema:



Database design and optimizations:

We deliberately populated the databases with strict constraints, focusing on highly specific data subsets. This strategic choice ensures we can derive meaningful insights from the data, even though it is significantly limited in scope.

Data insertion is being handled as such:
For each movie we're inserting into the database, we also insert its 5 main actors (as the same actor can be the main actor in more than one movie, it will be at most 5 actors), its genres and the relations between them (movie-actor, movie-genre).
This way of insertion makes sure that all data in the database has some relations to other data, enforcing interesting insights for the user.

All attributes in all tables were based on API requests but were tweaked to efficient size (e.g. movie_id INT was changed to SMALLINT).

Also, we chose that all of our queries would use user input, making the application more interactive and tailored in use for the user.

Tables:

movies table:

This table functions as the "main" table, holding valuable data for a movie producer regarding few movies' main aspects.

We chose to implement an index on the "title" attribute, using it in our "title helper" query, making it much easier for the server to search the specific keyword entered by the user.

We also chose to add another index "index_budget" as for one of our queries ("runtime helper") is using the movies budget to determine which movies to be included in the calculation, this index helps the query to run efficiently, for the small expanse of a single INT attribute index in memory.

actors table:

This table is being populated with the cast members of each of the movies that were inserted to the movies table, with a limit of 5 cast members per movie and duplicate being ignored (though duplicated are still inserted to the movies_actors table).

We chose to add another index on actors.name, as we often use it to implement our "actor helper" query, making it run much faster while traversing the large table.

movies_actors table:

Derived from a many-to-many relation between movies and actors, this table uses a primary key of double attributes (movie_id, actor_id), and foreign keys in order to enforce data integrity. Also, because of the use of foreign keys, our database also implemented a second index (native system behavior) on actor_id, to be able to enforce the foreign key resection efficiently (and not on movie_id because the PRIMARY key is already sorted based on it).

genres table:

Similarly to actors table, each of the genres that are inserted must be included in one of the movies in the movies table. Here, we did not enforce a limit like we did for actors because of the different nature of movies genres compared to cast, also many more duplicates.

movies_genres table:

Similarly to movies_actors table, automatically generated index on genre_id and a PRIMARY index on the PRIMARY key (movie_id, genre_id).

Queries implementation:

Query 1 – "Genre Helper":

```
SELECT movies.title, (movies.revenue - movies.budget) AS profit
FROM movies, genres, movies_genres
WHERE genres.name = %s
    AND genres.genre_id = movies_genres.genre_id
    AND movies_genres.movie_id = movies.movies_id
ORDER BY profit DESC
LIMIT 5;
```

This query gets a genre as an input and returns the 5 most profitable movies of this specific genre. The query uses 3 tables (genres, movies, movies_genres) to process the movie information after the user input of a genre name.
After choosing a genre, the query calculates for each movie in this genre its final profit, and return the 5 most profitable in order.

Query 2 – "Actor Helper":

```
SELECT AVG(movies.popularity) AS avg_popularity
FROM movies
JOIN movies_actors ON movies.movie_id = movies_actors.movie_id
JOIN actors ON actors.actor_id = movies_actors.actor_id
WHERE MATCH(actors.name) AGAINST(%s IN BOOLEAN MODE)
    AND (
        CASE
        WHEN INSTR(%s, ' ') > 0 THEN TRUE
        ELSE actors.name REGEXP CONCAT('(^|[[:space:]])', %s, '([[:space:]]|$)')
        END
  );
```

This Full-text query gets an actor name as input, and returns the average popularity of movies that he starred in.

The query handles the input actor name as follows:

1. The query takes the input as a whole, meaning if "Jam" was searched no actors with the name "James" would be returned.
2. We have a case statement for either just first or last name, or a full name input.
    a. If the user input only first or last name, the query would search for all actors containing this name in their first or last name.
    b. If the user input 2 words (known by identifying a " " char in the case), the query would only check for actors with this specific full name.

After the list of actors was found, the query returns the average popularity of the films that this actor/s has played in, thus helping the producer find a "popular actor".
The query uses 3 tables (actors, movies, movies_actors) and a full-text index "name", in order to process the actor name, then find the links to the specific movies, and lastly calculate the average popularity.

Query 3 – "Title Helper":

```
SELECT AVG(movies.revenue - movies.budget) AS avg_revenue
FROM movies
WHERE (
  CASE
    WHEN INSTR(%s, ' ') > 0 THEN movies.title = %s
    ELSE movies.title REGEXP CONCAT('(^|[[:space:]])', %s, '([[:space:]]|$)')
  END
);
```

This query gets a keyword as an input and returns the average income (revenue-budget) of movies containing this keyword in their name.
The query would only check for full words, if the user inputs the word "War", it will not return "Star Wars" as "War"!= "Wars" not treated the same as a full word.
(Specifically Star Wars is written "Star Wars**:** …" in the TMDB)
This query only uses the table movies and the added full-text index "title" to traverse the movies names more efficiently and the movie table columns (budget, revenue) to calculate the average income with the desired keyword.

Query. 4 – "Runtime Helper":

```
SELECT AVG(movies.runtime * (movies.revenue / movies.budget)) /
NULLIF(AVG(movies.revenue / movies.budget), 0) AS best_runtime
FROM movies
WHERE movies.budget BETWEEN %s AND %s
```

This query gets an input of a desired budget from the user, and returns best runtime for this movie under the following logic:

1. The query takes the income budget and filters out (using idx_budget on budget column) only movies with a similar budget (10% above and below) assisted by an external logic to set the upper and lower bound used later in the query.
   *we chose to implement it like this in order to only return movies with similar asked budget, as for this, if the user inputs a very big or very small budget – it might not get results at all (no other movies in the asked range), also, some data is missing in TDMB (budget 0$ for some movies), therefore a user input of 0$ would make no real answer, so we decided to not include 0$ budget in the answer.
2. After this list of movies has been found, weight each movie based on their revenue/budget ratio and multiply it by the movie runtime.
3. At last, the query normalizes the data to receive the correct best runtime.

$$\frac{\left(\sum_{n=movie}^{all\ movies} \frac{(\text{total } revenue)}{(\text{total } budget)} \cdot \text{length}\right)}{\sum_{n=movie}^{all\ movies} \frac{(\text{total } revenue)}{(\text{total } budget)}}$$

This query uses only the movies table and the index "idx_budget" on budget column and makes a complex mathematical computation on its attributes.
We decided to implement the index for the query to run more efficiently – instead of checking for matching movies budget and going through all over the movies, it would quickly find the movies in the budget range.

Query 5 – "Release date Helper":

```
SELECT DAY(release_date) AS day, AVG(popularity)
FROM movies
WHERE MONTH(release_date) = %s
GROUP BY DAY(release_date)
```

This query gets a month as an input and returns the average popularity of movies for each day of this month in a graph, helping the producer to visually see the most likely best day in the month to release the movie for it to be popular.
The query only uses the movies table, using its release date column in order to GROUP BY the dates, and using the popularity column to calculate the best day to release the movie.

Code structure and API workflow:

Our program relies on 4 important script files:

create_db_script:

Functionality:

Table Creation: Defines and creates tables (`movies`, `genres`, `movies_genres`, `actors`, `movies_actors`) if they don't already exist.

Index Creation: Creates an index on the `budget` column of the `movies` table.

Table Deletion: Deletes all tables, disabling foreign key checks to ensure all tables can be dropped regardless of dependencies

api_data_retrieve:

Functionality:

API Data Fetching: Fetches data from the TMDB (The Movie Database) API. Specifically, we take 300 pages of the top-rated movies, and for each movie use an API call to find all its details (budget, revenue, genres, etc.). For each such movie we use another API call to find its credits and take the top 5 main actors in the cast.

Data Insertion: Inserts fetched data into the MySQL database, including movies, actors, and genres.

Error Handling: Handles errors during data fetching and insertion. Even though we insert entries into the tables using the keyword IGNORE, we still check if an error of this type is detected and skip those cases, just in case. For every other error we throw an error and terminate.

queries_db_script:

Functionality:

Database Queries: Contains various queries on the MySQL database that retrieve specific information. Those queries will be executes from the **queries_execution** python file.

Data Formatting: Formats query results using the prettytable library for better readability.

queries_execution:

Functionality:

Database Connection: Establishes a connection to a MySQL database.

Main function: Serves as the main function and includes a CLI for query investigation.

User Input Handling: Prompts the user for input values such as budget and month and handles wrong inputs.

Query Execution: Executes predefined queries based on user input.

Data Visualization: Displays query results in a readable format using prettytable and visualizes data using `matplotlib`.