CinemaCentral.com

DATA :

- The data was taken from the Kaggle website.
- It is merged into one CSV (merged_tmdb_data.csv) file from 2 different CSV files with ~9000 rows.
- Most of the data was structured however we needed to alter some columns like keyword, cast, revenue, and budget so we could insert them into the tables.

Database schema structure and reasoning:

Movies_data:

Purpose: Central repository for comprehensive movie details.

- movie_id (Primary-Key): `Unique identifier for each movie.`
- title_x: `Title of the movie.`
- overview: `Brief description or summary of the movie.`
- runtime: `Duration of the movie in minutes.`
- rating: `Rating of the movie.`
- keywords: `Keywords associated with the movie.`
- popularity: `Popularity score of the movie (watched).`
- release_date: `Release date of the movie.`

- Reasoning : This table provides a foundation for various movie-related queries, enabling easy retrieval of movie-specific data without the need for complex joins. In addition, it allows efficient analysis, categorization, and comparison of movies based on various criteria.

genres_table:

Purpose: Stores distinct movie genres.

- genres_id (Primary-Key): `Unique identifier for each genre`.
- genre_name: `Name of the genre.`

– Reasoning : This table centralizes genre information, ensuring consistency in genre names across the database, which enables efficient filtering and retrieval of movies based on genre preferences.

movies_genres:

Purpose: Manages the many-to-many relationship between movies and genres.

- movie_id (Foreign Key): References the `movie_id` in the `movies_data` table.
- genres_id (Foreign Key): References the `genres_id` in the `genres_table`.

- Reasoning : This table resolves the complex relationship between movies and genres, allowing a movie to belong to multiple genres and vice versa. It enables efficient genre-based searches and filtering of movies.

actors:

Purpose: Repository for actor details

- actor_id (Primary Key): `Unique identifier for each actor.`
- actor_name: `Name of the actor.`

– Reasoning : The table centralizes information about actors, including unique identifiers and names. It supports actor-related queries and analysis, such as tracking an actor's involvement in various movies and assessing their popularity. And enables efficient retrieval and management of actor data across the database.

movie_actor:

Purpose: Manages the many-to-many relationship between movies and actors.

- movie_id (Foreign Key): References the `movie_id` in the `movies_data` table.
- actor_id (Foreign Key): References the `actor_id` in the `actors` table.

- Reasoning : This table enables efficient queries related to actors' contributions to movies and vice versa. And ensures data integrity by enforcing referential integrity through foreign key constraints.

profit:

Purpose: Tracks financial information related to movies.

- movie_id (Primary Key, Foreign Key): References the `movie_id` in the `movies_data` table.
- production_company: `Name of the production company.`
- revenue: `Revenue generated by the movie.`
- budget: `Budget allocated for the movie.`

- Reasoning : This table stores revenue, budget, and production company data for each movie, facilitating analysis of movie profitability and financial performance. By doing so it supports decision-making processes related to movie investments, budget allocations, and revenue forecasting.

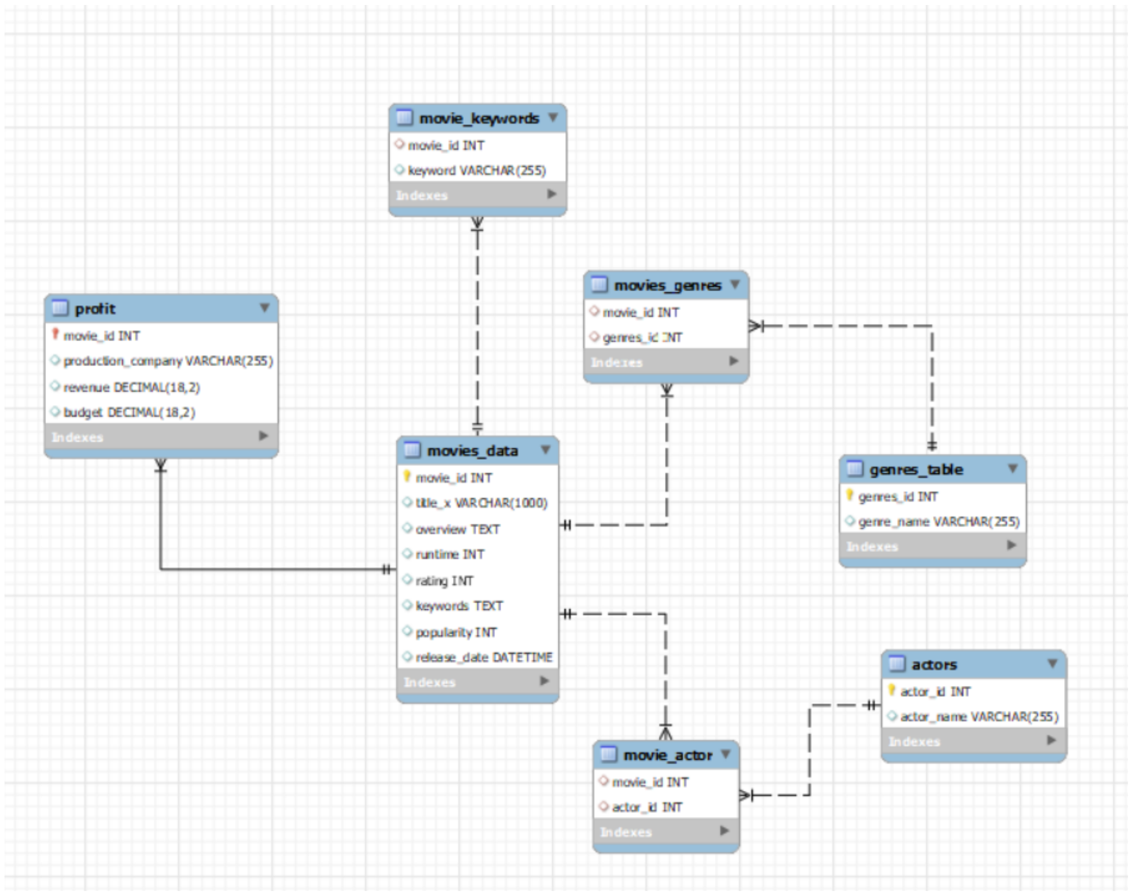
movie_keywords:

Purpose: Stores keywords associated with movies.

- movie_id (Foreign Key): References the `movie_id` in the `movies_data` table.
- keyword: `Keywords associated with the movie.`

- Reasoning : This table enhances discoverability and recommendation features by tagging movies with relevant keywords. It facilitates keyword-based searches and filtering of movies based on specific topics or themes.

EER Diagram:


movie_keywords
movie_id INT
keyword VARCHAR(255)
Indexes
movies_genres
movie_id INT
genres_id INT
Indexes
profit
movie_id INT
production_company VARCHAR(255)
revenue DECIMAL(18,2)
budget DECIMAL(18,2)
Indexes
movies_data
movie_id INT
title_x VARCHAR(1000)
overview TEXT
runtime INT
rating INT
keywords TEXT
popularity INT
release_date DATETIME
Indexes
genres_table
genres_id INT
genre_name VARCHAR(255)
Indexes
actors
actor_id INT
actor_name VARCHAR(255)
Indexes
movie_actor
movie_id INT
actor_id INT
Indexes

Database optimizations:

These indexes we added allow the database to efficiently execute common operations like sorting, filtering, and joining, resulting in faster query response times and improved overall database performance. This optimization ensures a smoother user experience and better scalability, especially as the volume of data grows.
We added indexes on `movie_id` in `movies_data` table, `genre_name` in `genres_table`, and `actor_id` in `actors` table, which provides several benefits:

- **Improved Query Performance :** Indexes allow the database to quickly locate rows based on the indexed columns. For example, when querying movies by `movie_id`, genres by `genre_name`, or actors by `actor_id`, the database can efficiently find the relevant rows without scanning the entire table.

- **Faster Joins:** Indexes on columns used in JOIN conditions, such as `movie_id`, `genre_name`, and `actor_id`, speed up join operations. When joining tables, the database can utilize these indexes to quickly match rows from different tables based on the indexed columns, reducing the time required to perform the join.

- **Optimized WHERE Clause Filtering:** Indexes enable efficient filtering of rows based on the indexed columns in WHERE clauses. For example, when querying movies based on `movie_id`, genres based on `genre_name`, or actors based on `actor_id`, the database can quickly identify and retrieve the relevant rows without scanning the entire table.

Detail 5 main Queries :

1. **Query 1:** Show Top 10 Movies Based on Rating
   - Purpose: Display a list of the top 10 movies based on their rating.
   - Optimization Strategy: Order the movies by rating in descending order to prioritize highly-rated movies. Limit the result set to 10 to focus on the top-rated movies only.
   - Database Design Support: The movies_data table contains movie information, including ratings, allowing efficient sorting based on the rating column.

Run example:

movie_name
Star Wars
Finding Nemo
American Beauty
Pirates of the Caribbean: The Curse of the Black Pearl
Kill Bill: Vol. 1
Apocalypse Now
Eternal Sunshine of the Spotless Mind
Memento
American History X
Million Dollar Baby

2. **Query 2:** Show Information about a Movie When Clicking the Name

- Purpose: Retrieve detailed information about a specific movie based on its name.
- Optimization Strategy : Filter the movies based on the provided movie name to ensure only the relevant movie's information is returned.
- Database Design Support : Indexing on movie_id in movies_data table allows for efficient retrieval of movie details, supporting the query's purpose.

Run example on “Winnie the pooh”:

movie_id	title_x	overview	runtime	rating	keywords	popularity	release_date
51162	Winnie the Pooh	During an ordinary day in Hundred Acre Wood, Winnie the Pooh sets out to find some honey…	63	7	["owl", "tiger", "aftercreditsstinger", "duringcreditsstinger"]	19	2011-04-13

3. **Query 9:** Search a Movie by Keywords

- Purpose: Search for movies based on keywords associated with them.
- Optimization Strategy: Utilize full-text search capabilities to match keywords against the movie's keyword list efficiently. Use BOOLEAN MODE search with wildcard characters for flexible keyword matching.
- Database Design Support: The `movie_keywords` table links movies to keywords, enabling efficient keyword-based searches.

Run example on “magi” (limited to 5):

title_x
Tangled
Harry Potter and the Half-Blood Prince
Oz: The Great and Powerful
Green Lantern
Snow White and the Huntsman

4. **Query 5:** Show Top 10 Movies in a Given Genre

- Purpose: Display the top 10 movies within a specified genre based on their popularity (most watched).
- Optimization Strategy: Use a subquery to select movies within the specified genre, order them by popularity, and limit the result set to 10. This ensures only the top-rated movies within the genre are returned.
- Database Design Support : The movies_genres table links movies to genres, facilitating genre-based filtering. Indexing on the genre_name column in the genres_table enhances the efficiency of genre-specific queries.

Run example on “Romace”:

movie_id	movie_name	popularity
8966	Twilight	127
102651	Maleficent	111
24021	The Twilight Saga: Eclipse	107
150689	Cinderella	101
597	Titanic	100
216015	Fifty Shades of Grey	99
18239	The Twilight Saga: New Moon	95
812	Aladdin	93
350	The Devil Wears Prada	84
293863	The Age of Adaline	82

5. **Query 6:** Show Top 10 Actors in a Specific Genre

- Purpose: Retrieve the top 10 actors who have appeared in the most movies within a specific genre.
- Optimization Strategy : Join the actors, movie_actor, movies_genres, and genres_table tables to count the number of movies each actor has appeared in within the specified genre. Order the actors by movie count and limit the result set to 10.
- Database Design Support: The database design includes tables for actors, movies, and genres, facilitating efficient actor-counting and genre-based filtering. Indexing on the actor_id column in the actors table supports efficient actor-based queries.

Run example on “Romace”:

actor_name	movie_count
Jason Bateman	9
Anna Kendrick	7
Amy Adams	6
James Marsden	6
Paul Rudd	6
Rebel Wilson	6
Kate Hudson	6
Anne Hathaway	6
Jennifer Aniston	6
Owen Wilson	5

Code Structure and general Flow:

The code is divided to 4 files :

1. create_db_script.py: Handles table creation in the database.
2. api_data_retrieve.py: Manages data insertion from CSV files into corresponding database tables. Also includes helper functions for formatting CSV data to match table structures.
3. queries_db_script.py: Contains query functions necessary for the website's functionality.
4. queries_execution.py: Orchestrates the process from table creation to query execution, including examples.

The website flow: when executing the queries_execution.py file, which follows these steps:

1. Creation of tables: The script creates the necessary tables in the database.
2. Data population: The tables are then populated with relevant data.
3. Example queries execution: Finally, the script executes example queries and prints the outputs based on the data in the tables.