**System Documentation**

**Database Schema Structure**

The database schema is structured to store and manage movie-related data efficiently while ensuring accuracy and supporting advanced queries. It is organized into five main tables, each with a specific role in maintaining structured and accessible information.

**1. Movies Table**

This table contains detailed information about each movie in the database, including:

* **movie\_id** – Unique identifier for each movie (Primary Key)
* **title, tagline, overview** – General descriptive information
* **budget, revenue, runtime, release\_date** – Financial and runtime details
* **original\_language, popularity, vote\_average, vote\_count** – Metadata related to audience engagement

**Indices:**

* Indexes on **revenue** and **popularity** improve query performance for sorting and filtering.

**2. Genres Table**

This table defines movie genres, allowing categorization of films by type. It includes:

* **genre\_id** – Unique identifier for each genre (Primary Key)
* **name** – Genre name

**3. Movie Genres Table**

This table connects movies and genres, enabling a many-to-many relationship between them.

* **movie\_id** – References the **Movies** table (Foreign Key)
* **genre\_id** – References the **Genres** table (Foreign Key)

**4. Production Companies Table**

This table lists production companies associated with movie creation.

* **company\_id** – Unique identifier for each production company (Primary Key)
* **name** – Name of the production company

**5. Movie Companies Table**

A junction table that creates a **many-to-many relationship** between movies and production companies.

* **movie\_id** – References the **Movies** table (Foreign Key)
* **company\_id** – References the **Production Companies** table (Foreign Key)

**Reasoning for the Chosen Database Design**

The database design was crafted to balance scalability, efficiency, and data integrity, ensuring optimal performance even with a large dataset of over 5,000 records.

**Key Considerations Behind the Design**

To ensure efficiency, maintainability, and scalability, the database schema was designed with the following principles in mind:

**1. Normalization for Data Integrity**

The schema follows **normalization best practices** to reduce redundancy and maintain consistency:

* Genres and production companies are stored in **separate tables**, rather than embedded within the movies table.
* This setup makes updates more efficient—if a genre's name needs to be changed, it only requires modifying one record in the **Genres** table instead of updating multiple movie entries.

**2. Many-to-Many Relationships**

Because a movie can have multiple genres and be linked to different production companies, these relationships are handled using junction tables (movie\_genres and movie\_companies).

* This structure allows for **efficient queries**, such as retrieving all movies in a specific genre or listing all genres assigned to a movie.

**3. Separate Tables for Lookup Values**

By storing data such as genres and production companies in **dedicated tables**, the database ensures **data consistency** and enables **advanced queries**, like filtering movies by production company or grouping results by genre.

**4. Scalability and Flexibility**

The schema is optimized to handle **large datasets and complex queries** without performance degradation:

* Data is structured across multiple tables, preventing bottlenecks.
* Proper indexing ensures queries remain efficient as the dataset grows.

**5. Drawbacks of Alternative Designs**

Alternative approaches, such as denormalization, would introduce **several challenges**:

* **Storing genres as comma-separated values** within the movies table would increase redundancy, complicate updates, and slow down queries.
* **Avoiding junction tables** would require complex string parsing, making it harder to filter or group data effectively.

**Database Optimizations**

To enhance database performance and ensure responsiveness, several optimizations have been implemented. These improvements focus on speeding up queries, maintaining data integrity, and supporting scalability as the dataset grows.

**1. Numeric Indices for Performance**

To improve query efficiency, indices have been applied to key numerical fields:

* **Revenue and popularity** columns are indexed to enable fast sorting and filtering.
* This optimization allows for quick retrieval of high-revenue movies or trending films without scanning the entire dataset.

**2. Efficient Data Ingestion**

The data insertion process in 'api\_data\_retrieve' includes mechanisms to maintain **data integrity** and reduce errors:

* **INSERT IGNORE** prevents duplicate records from being added.
* **Data validation** checks ensure that only valid IDs and structured information are inserted into the database.
* These steps help maintain a clean dataset while minimizing potential runtime issues.

**3. Foreign Keys and Referential Integrity**

To enforce **data consistency**, foreign key constraints are applied in junction tables such as 'movie\_genres' and 'movie\_companies'.

* These constraints prevent orphaned records and maintain relationships between movies, genres, and production companies.

**4. Optimized Query Structure**

Queries are structured to take full advantage of indexing and normalization, ensuring optimal performance:

* **GROUP BY queries** efficiently count movies by genre using indexed columns.
* **EXISTS conditions** in 'movie\_genres' help streamline queries, reducing unnecessary joins while verifying relationships.

**Detail of the Five Main Queries**

1. **Search by Movie Description (overview)**:
   * **Purpose**: Allows users to find movies containing specific words or phrases in their description.
   * **Query**:

SELECT \*

FROM movies

WHERE overview LIKE CONCAT('%', %s, '%')

1. **Search by Movie Tagline**:
   * **Purpose**: Enables users to search for movies based on their promotional tagline.
   * **Query**:

SELECT \*

FROM movies

WHERE tagline LIKE CONCAT('%', %s, '%')

1. **Movies Count per Genre**:
   * **Purpose**: Retrieves the number of movies categorized under each genre, ordered by the highest count.
   * **Query**:

SELECT genres.name, COUNT(movie\_genres.movie\_id) AS count\_movies

FROM genres

JOIN movie\_genres ON genres.genre\_id = movie\_genres.genre\_id

GROUP BY genres.genre\_id

ORDER BY count\_movies DESC

1. **Highest Revenue Movie**:
   * **Purpose**: Identifies the movie that generated the highest revenue.
   * **Query**:

SELECT title, release\_date, revenue

FROM movies

WHERE revenue = (SELECT MAX(revenue) FROM movies)

1. **Movies in a Specific Genre** :
   * **Purpose**: Lists all movies associated with a specific genre (e.g., Action with genre\_id = 28).
   * **Query**:

SELECT title

FROM movies

WHERE EXISTS (

SELECT \*

FROM movie\_genres

WHERE movie\_genres.movie\_id = movies.movie\_id

AND movie\_genres.genre\_id = 28

)

**Code Structure and API Usage**

1. **Code Structure**:
   * **create\_db\_script.py**: Handles the creation of the database schema and tables.
   * **api\_data\_retrieve.py**: Handles the automated import of data into the database from a CSV file.
   * **queries\_db\_script.py**: Contains functions for executing SQL queries.
   * **queries\_execution.py**: Demonstrates how to use the query functions and retrieve results.
2. **External Libraries**:
   * **Pandas**: Used for handling and processing the CSV file.
   * **tqdm**: Provides progress bars for data processing.
   * **mysql-connector**: Manages connections and queries to the MySQL database.
3. **API Usage**:
   * While the project primarily uses a CSV file, it could be extended to retrieve data dynamically from APIs for additional functionality.