**System Documentation**

**Database Schema Structure**

The database schema is designed to manage movie-related data efficiently while maintaining flexibility for complex queries and ensuring data integrity. It consists of five main tables and leverages normalized structures to organize data.

1. **Movies**:
   * Stores all movie-related details, including:
     + movie\_id (Primary Key)
     + title, tagline, overview
     + budget, revenue, runtime, release\_date
     + original\_language, popularity, vote\_average, vote\_count
   * Indices:
     + Full-text index on overview and tagline for optimized text searches.
     + Additional indices on revenue and popularity for efficient sorting and filtering.
2. **Genres**:
   * Represents movie genres, with:
     + genre\_id (Primary Key)
     + name (Genre name).
3. **Movie Genres**:
   * Junction table connecting movies and genres to represent many-to-many relationships.
   * Columns:
     + movie\_id (Foreign Key referencing movies)
     + genre\_id (Foreign Key referencing genres)
4. **Production Companies**:
   * Lists companies involved in movie production:
     + company\_id (Primary Key)
     + name (Company name).
5. **Movie Companies**:
   * Junction table linking movies and production companies to represent many-to-many relationships.
   * Columns:
     + movie\_id (Foreign Key referencing movies)
     + company\_id (Foreign Key referencing production\_companies).

**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:

1. **Normalization for Data Integrity**:
   * The schema adheres to normalization principles to avoid redundancy and ensure consistency. For instance:
     + Genres and production companies are stored as separate tables instead of being embedded in the movies table.
   * This structure simplifies updates and reduces storage requirements. For example, a genre's name needs to be updated only once in the genres table rather than in multiple movie records.
2. **Many-to-Many Relationships**:
   * Real-world relationships such as movies having multiple genres or being associated with multiple production companies are represented through junction tables (movie\_genres and movie\_companies).
   * This structure enables efficient queries, such as finding all movies in a genre or all genres of a specific movie.
3. **Separate Tables for Lookup Values**:
   * Data like genres and production companies are stored in their respective tables to maintain consistency. This design also allows for advanced queries, such as finding all movies produced by a specific company or grouped by genre.
4. **Scalability and Flexibility**:
   * The schema is designed to handle large datasets and complex queries without performance degradation. For example, separating data into distinct tables with proper indexing ensures that queries remain efficient even as the number of records grows.
5. **Drawbacks of Alternative Designs**:
   * A denormalized schema (e.g., storing genres as comma-separated values in the movies table) would increase redundancy, complicate updates, and make querying inefficient.
   * Avoiding junction tables would require complex string parsing and limit the ability to filter or group by relationships.

**Database Optimizations**

To maximize the database's performance and responsiveness, several optimizations were applied. These focus on improving query speeds and ensuring scalability for future growth.

* **Full-Text Indices**:
  + Applied on the overview and tagline columns in the movies table.
  + Purpose:
    - Enables efficient free-text searches, which are commonly used to find movies based on descriptions or promotional taglines.
    - Allows the use of MATCH...AGAINST, which is significantly faster than LIKE with wildcards, especially on large datasets.
  + Without these indices, text searches would require a full table scan, which is resource-intensive for large datasets.
* **Numeric Indices**:
  + Applied on revenue and popularity columns in the movies table.
  + Purpose:
    - Optimizes queries that involve sorting or filtering based on revenue or popularity, such as finding the highest-earning movie.
  + Without these indices, the database would need to scan all rows to compute the highest or lowest values, leading to slower performance.
* **Efficient Data Ingestion**:
  + The data insertion process in api\_data\_retrieve.py uses efficient techniques such as:
    - INSERT IGNORE to prevent duplicate entries.
    - Validation of data (e.g., checking if IDs are valid before insertion).
  + These techniques ensure clean and consistent data while minimizing runtime errors during bulk data loads.
* **Foreign Keys and Referential Integrity**:
  + Foreign key constraints are used in junction tables to maintain relationships between entities (e.g., movie\_genres and movies).
  + These constraints ensure that relationships are valid, preventing orphaned records.
* **Scalable Query Design**:
  + Queries are designed to take advantage of the normalized schema and indices. For example:
    - GROUP BY queries efficiently count movies per genre using indexed columns.
    - EXISTS checks in movie\_genres avoid unnecessary joins when 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 MATCH(overview) AGAINST (%s IN NATURAL LANGUAGE MODE)

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

SELECT \*

FROM movies

WHERE MATCH(tagline) AGAINST (%s IN NATURAL LANGUAGE MODE)

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 (e.g., Action)**:
   * **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**: Automates the process of populating the database with data 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.