Movie Database Management System - System Documentation

1. Database Schema

The database consists of the following tables:

- genres (id, genre_name): Stores unique movie genres.
- directors (id, director_name): Stores movie directors.
- actors (id, actor_name, biography): Stores actor details with full-text search on biographies.
- movies (id, title, description, genre_id, director_id): Stores movies with references to genres and directors.
- film_actor (id, movie_id, actor_id): Many-to-many relationship between movies and actors.

Database Design Rationale

- Normalization (3NF): Ensures minimal redundancy and data integrity.
- Primary Keys:
  - genres.id - Uniquely identifies each genre.
  - directors.id - Uniquely identifies each director.
  - actors.id - Uniquely identifies each actor.
  - movies.id - Uniquely identifies each movie.
  - film_actor.id - Ensures uniqueness of relationships between movies and actors.
- Foreign Keys:
  - movies.genre_id → References genres.id, ensuring movies are linked to valid genres.
  - movies.director_id → References directors.id, linking each movie to a valid director.
  - film_actor.movie_id → References movies.id, associating actors with movies.
  - film_actor.actor_id → References actors.id, ensuring only existing actors are linked to movies.
- Many-to-Many Relationships: film_actor bridges movies and actors efficiently.
- Alternative Design Considerations:
  - Denormalization: Could improve read speed but increase data redundancy.
  - Single Table for Movies & People: Would simplify structure but reduce efficiency for queries.

2. Database Optimizations

- **Indexes:**
    - **Primary Keys & Foreign Keys: Speed up joins and lookups.**
    - **Full-Text Indexes: Optimize search in movies.title, movies.description, and actors.biography.**
    - **Indexes on genre_id and director_id: Enhance filtering speed.**
- **Query Optimization: Uses efficient joins and subqueries to reduce unnecessary scans.**

3. Main Queries & Database Support

1. **Movie Search:**
    - **Uses MATCH(title, description) AGAINST() for full-text search.**
    - **Indexing supports fast retrieval, allowing quick searching of movie titles and descriptions based on user queries.**
2. **Actor Search:**
    - **Uses full-text search in biography to find relevant actors.**
    - **This allows users to search for actors based on keywords such as "Oscar winner" to retrieve actors with relevant biographical details.**
3. **Most Popular Genre:**
    - **Aggregates movies per genre_id, utilizing indexes for fast computation.**
    - **This helps identify which genres have the most movies stored in the database, providing insights into trends.**
4. **Actor Collaborations:**
    - **Self-join on film_actor enables finding actor pairs who co-starred in multiple movies.**
    - **This allows analysts to find frequent collaborators in the movie industry, identifying actor pairs who appear together often.**
5. **Directors with Diverse Genres:**
    - **Uses COUNT(DISTINCT genre_id) HAVING >= 3 to identify versatile directors.**
    - **This helps highlight directors who have worked across multiple genres, showcasing their flexibility in filmmaking.**

4. Code Structure & API Usage

- **create_db_script.py - Initializes the database schema.**
- **queries_db_script.py - Contains predefined SQL queries.**
- **api_data_retrieve.py - Fetches and updates movie-related data.**

- **populate_db.py - Fetches external movie data and populates the database.**
- **API Integration:**
    - **Retrieves movie data, genres, actors, and directors.**
    - **Handles rate limits and errors.**