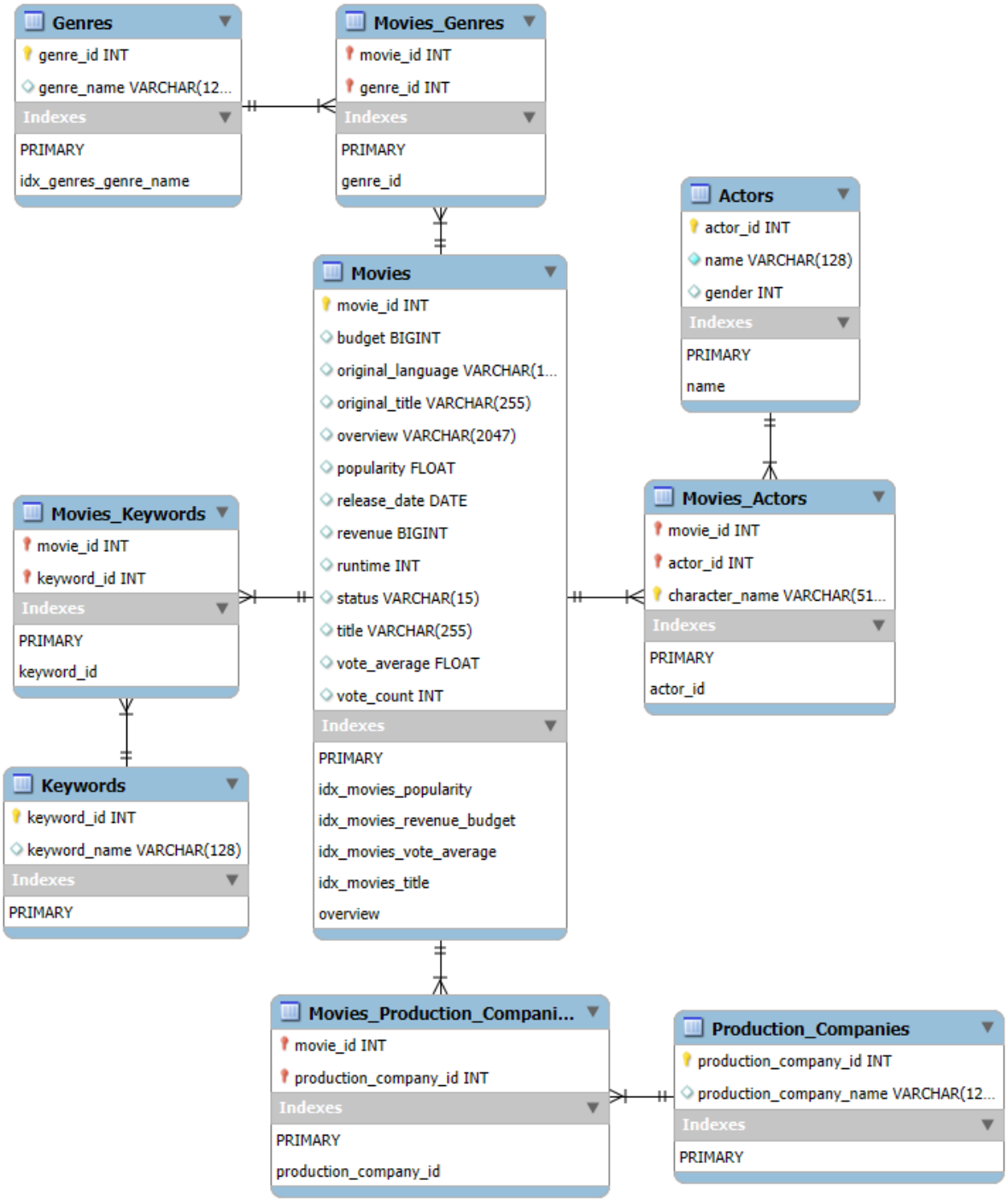
System Documentation

Contents

1. Database Schema Structure 2
2. Database Design and Justification 4
3. Database Optimizations, Index Usage 6
   3.1 Database Optimizations 6
   3.2 Index Usage 7
4. Main Queries and Database Support 9
   Query 1: Search for Movies by Overview Keywords 9
   Query 2: Find Actor Dynasties by Last Name 9
   Query 3: Find Top 5 Most Profitable Movies in a Genre 9
   Query 4: Find Top 10 High-Rated Actors in a Genre 10
   Query 5: Find Top 5 Production Companies by Revenue 10
   Query 6: Suggest Related Movies Based on Shared Keywords 10
5. Code Structure and API Usage 12
   5.1 Code Structure 12
   5.2 API Usage: Gathering Data from Kaggle 13

1. Database Schema Structure



The database schema consists of multiple interrelated tables to store and manage movie-related data efficiently. The tables include:

- **Movies**: Stores information about movies, including their budget, revenue, popularity, release date, and more.

  Movies(movie_id, budget, original_language, original_title, overview, popularity, release_date, revenue, runtime, status, title, vote_average, vote_count)

- **Genres**: Stores different genres of movies.

  Genres(genre_id, genre_name)

- **Movies_Genres**: A many-to-many relationship table connecting movies with their genres.

  Movies_Genres(movie_id, genre_id)

  FKs;
  Movies_Genres.movie_id → Movies.movie_id
  Movies_Genres.genre_id → Genres.genre_id

- **Keywords**: Stores movie keywords (tags).

  Keywords(keyword_id, keyword_name)

- **Movies_Keywords**: A many-to-many relationship table connecting movies with keywords.

  Movies_Keywords(movie_id, keyword_id)

  FKs;
  Movies_Keywords.movie_id → Movies.movie_id
  Movies_Keywords.keyword_id → Keywords.keyword_id

- **Production_Companies**: Stores information about movie production companies.

  Production_Companies(production_company_id, production_company_name)

- **Movies_Production_Companies**: A many-to-many relationship table connecting movies with production companies.

  Movies_Production_Companies(movie_id, production_company_id)

  FKs;
  Movies_Production_Companies.movie_id → Movies.movie_id
  Movies_Production_Companies.production_company_id →
  Production_Companies.production_company_id

- **Actors**: Stores details about actors, including their name and gender.

  Actors(actor_id, name, gender)

  Database constraints:
  ```
  name VARCHAR(128) NOT NULL
  gender INT CHECK (gender IN (0, 1, 2))
  ```

- **Movies_Actors**: A many-to-many relationship table connecting movies with actors, including character names.

  Movies_Actors(movie_id, actor_id, character_name)

  FKs;
  Movies_Actors.movie_id → Movies.movie_id
  Movies_Actors.actor_id → Actors.actor_id

In addition to indexes on PKs and FKs that are created automatically in mysql, following indexes were defined;

Full Text Indexes;

On Movies(overview), Actors(name).

B-Tree Indexes;

On Movies(popularity), Genres(genre_name), Movies(vote_average), Movies(title),
composite index on Movies(revenue, budget).

2. Database Design and Justification

The chosen design adheres to Third Normal Form (3NF) principles to minimize data redundancy and improve maintainability.

Advantages of the Chosen Design:

- **Efficiency**: The many-to-many relationship tables avoid data duplication in the Movies, Genres, Keywords, and Actors tables.
- **Index Optimization**: Indexes improve search efficiency for large datasets (see next).
- **Data Integrity**: Foreign keys enforce referential integrity.

Alternative Designs Considered and Drawbacks:

The simplest design is a **single denormalized table**: This approach would have required redundant storage of genres, actors, and production companies within the same row, leading to inefficiencies and update anomalies.

More practical alternatives include;

1. Star Schema

Star schema consists of:

- A central "fact" table (Movies) containing quantifiable movie-related data (budget, revenue, popularity, etc.).
- Multiple "dimension" tables (Genres, Actors, Production Companies etc.) that contain descriptive attributes.

Schema Structure:

- Fact Table (Movies_Fact):
    - Contains movie_id (PK), budget, revenue, popularity, vote_average, etc.
    - Also stores foreign keys (genre_id, actor_id, production_company_id) linking to dimensions.
- Dimension Tables (Genres, Actors, Production_Companies, etc.):
    - Each contains a primary key (e.g., genre_id, actor_id) and descriptive attributes.

Advantages:

- Optimized for Analytical Queries: Aggregations on revenue, ratings, or movie performance are fast due to predefined relationships.
- Simplicity: The single fact table simplifies querying in analytics environments.

Drawbacks:

- Lack of Flexibility: Some movies belong to multiple genres, but in a star schema, one-to-many relationships aren't well supported.
- Data Duplication: If a movie has multiple actors or genres, the fact table needs multiple rows for the same movie, leading to redundancy.
- Update Anomalies: due to redundancy in the fact table when handling many-to-many relationships. a star schema stores multiple rows for the same movie in the fact table (e.g. one per each combination of genre, production company, actor etc.), adding new relationships (e.g. a new actor) requires duplicating all relevant rows.

For example, in order to add an actor to a movie cast, we have to add many new fact table rows for that movie; If the movie is associated with 3 genres, all 3 rows must be duplicated to include the new actor. This leads to data bloat, inefficiencies, and potential inconsistencies if an update is applied incorrectly.

2. Snowflake Schema

A snowflake schema is a variation of the star schema that breaks down dimension tables into smaller sub-tables, creating highly normalized dimensions.

Schema Structure:

- The fact table (Movies_Fact) stores factual movie data.
- Dimension tables (Genres, Actors, Production_Companies etc.) are further normalized:
    - Genres table is split into Genres (genre_id, genre_name) and Movies_Genres (mapping table).
    - Actors is split into Actors (actor_id, name) and Movies_Actors (mapping table).

Advantages (compared to 1):

- Reduces Data Redundancy: More normalized than a star schema, reducing duplicate data.

Drawbacks:

- More Complex Queries: Queries require more joins due to further normalization (more sub-tables), slowing down performance (e.g; Movies_Genres → Genres → Genre_Types (2 joins)). Optimized for storage savings but at the cost of more joins.

3. Database Optimizations, Index Usage

3.1 Database Optimizations

- Tables Were Normalized to Reduce Data Redundancy.
  Instead of storing genres, actors, and production companies directly in the Movies table, they were separated into dedicated tables (Genres, Actors, Production_Companies).
  This avoids repetitive storage of the same genre names, actor details, and production company names across multiple movie records.
- Many-to-Many Relationship Tables.
  Relationships between movies and genres, actors, and production companies were handled using mapping tables (Movies_Genres, Movies_Actors, Movies_Production_Companies).
  This enables efficient queries while maintaining data integrity.
- Foreign Keys for Referential Integrity.
  Table definitions include foreign keys, which help maintain data integrity by enforcing relationships between tables. This prevents orphaned records, ensures data consistency and avoids cascading update issues.
- Join on Indexed Columns.
  All joins are performed on columns that are PKs/FKs and thus are automatically indexed by mysql.
  Having index for columns used in a join helps speed it up.
- Prepares the Database for Scalability
  The structure supports scalable data expansion, meaning that new movies, genres, actors, and production companies can be added efficiently without disrupting existing relationships.

Several query optimizations were performed;

- Batch data insertion utilizes cursor.executemany().
  Batch processing was used for efficient data insertion.
- Prepared Statements for batch insertion queries.
  For data insertion queries in insert_data_row_by_row() from api_data_retrieve.py,
  one Connection cursor was created as a MySQLCursorPrepared cursor;

```
cursor = connection.cursor(prepared=True)
```

> A cursor instantiated from the `MySQLCursorPrepared` class works like this:
>
> - The first time you pass a statement to the cursor's `execute()` method, it prepares the statement. For subsequent invocations of `execute()`, the preparation phase is skipped if the statement is the same.

  from the mysql docs; This avoids re-parsing the same query, making execution of batch inserts faster.

  For all queries, a fully parametrized query string was passed to the cursor for execution, to avoid sql injection.
- Window Function was utilized in query_3 to avoid recalculating average profit in a subquery (also avoids adding a CTE);

```
AVG(m.revenue - m.budget) OVER (PARTITION BY g.genre_name) AS avg_profit
```

  calculates the average profit once per genre while keeping individual movie details.
- Full Text search in query_1 executed in a subquery to allow ordering by relevance without additional search;

```
FROM (
      SELECT movie_id, title, overview, popularity,
            MATCH(overview) AGAINST (%s IN NATURAL LANGUAGE MODE) AS relevance
      FROM Movies
      ) AS subquery
```

- CTE for precomputing average in query_4; ensures avg_vote is computed once and will be available for all rows.
- CTEs for query_6; first find movie_id, then find all keywords for that movie, only then evaluate main query, while these values are already available for join.

3.2 Index Usage

To enhance performance, various optimizations were implemented, primarily leveraging indexes and query structuring. Indexes are added to speed up specific queries by reducing the number of rows mysql needs to scan. Below is an explanation of how each index improves query performance.

Indexing Strategy

- **Full-Text Indexes**:

  Used to filter and rank results based on MATCH() AGAINST().

  - FULLTEXT(overview) on **Movies(overview)** for keyword-based searches, optimizes query_1 fulltext search.
  - FULLTEXT(name) on **Actors(name)** for actor name searches with partial matches (e.g. looking for all actors with same last name), optimizes query_2.

- **B-Tree Indexes**:

  Summary:

Index	Queries Improved	Where It Helps
idx_movies_popularity	query_1, query_6	Faster ranking by popularity
idx_movies_revenue_budget	query_3, query_5	Speeds up revenue & budget calculations
idx_genres_genre_name	query_3, query_4	Fast lookup of genre_name-based filtering
idx_movies_vote_average	query_4	Optimizes high-rated movie filtering
idx_movies_title	query_6	Faster movie title lookups

  Explanation:

  - idx_movies_popularity on **Movies(popularity)** for faster ranking queries.

    This index speeds up ranking queries where movies are sorted by popularity.
    It speeds up query_1 (also query_6); The filtered results are sorted by relevance DESC, popularity DESC.
    If two movies have the same relevance, they are sorted by popularity. With an index on popularity, mysql can retrieve sorted results faster, reducing sorting overhead.

    Example; Executing query_1 with a common keyword “father”, without the index took 76ms**;**

```
93          try:
94              execute_query(connection, query_1, *args: "father", max_width=128)
95              # execute_query(connection, query_2, "Skarsgard")
96              # execute_query(connection, query_3, "Comedy")
97              # execute_query(connection, query_4, "Drama")
98              # execute_query(connection, query_5)
```

```
EXPLAIN
-> Limit: 10 row(s)  (cost=712.61 rows=10) (actual time=76.239..76.324 rows=10 loops=1)
    -> Sort row IDs: relevance DESC, ...
```

    After adding the index it took 20ms;

```
-> Limit: 10 row(s)  (cost=712.61 rows=10) (actual time=20.165..20.245 rows=10 loops=1)
    -> Sort row IDs: relevance DESC, ...
```

- idx_movies_revenue_budget on **Movies(revenue, budget)** for profitability analysis.

  It speeds up query_3; Since we are computing (revenue - budget), indexing revenue and budget helps.

  Example; Executing query_3 with genre "Comedy" without the index took 2859ms**;**

```
-> Limit: 5 row(s)  (actual time=2859.361..2859.363 rows=5 loops=1)
    -> Sort: profit DESC  (actual time=2859.360..2859.361...
```

  After adding the index it took 989ms;

```
-> Limit: 5 row(s)  (actual time=989.055..989.057 rows=5 loops=1)
    -> Sort: profit DESC  (actual time=989.053..989.054 row...
```

  Also helps query_5, as when a query aggregates Movies.revenue using functions like SUM() or AVG(), mysql can scan this index instead of the full table, speeding up the operation.

- idx_genres_genre_name on **Genres(genre_name)** to optimize genre-related searches.

  Reduces lookup time for filtering movies by genre (WHERE g.genre_name = %s).
  It speeds up query_3 (also query_4);

  Example; Executing query_3 with genre "Comedy" without the index took (but with index on Movies(revenue, budget)) took 989ms.

  After further adding index on genre_name, the execution took 392ms;

```
-> Limit: 5 row(s)  (actual time=392.185..392.187 rows=5 loops=1)
    -> Sort: profit DESC  (actual time=392.184..392.185 row...
```

- idx_movies_vote_average on **Movies(vote_average)**

  Speeds up filtering of high-rated movies (WHERE vote_average > X).
  Prevents full table scans when searching for highly rated movies within a genre.
  Speeds up query _4.

  Example; Executing query_4 with genre "Drama" without the index took 4694ms**;**

```
EXPLAIN
-> Limit: 10 row(s)  (actual time=4694.465..4694.467 rows=10 loops=1)
    -> Sort: high_rated_movies DESC, limit input to 10 ...
```

  After adding the index it took 1501ms;

```
-> Limit: 10 row(s)  (actual time=1500.998..1501.000 rows=10 loops=1)
    -> Sort: high_rated_movies DESC, limit input to 10 ...
```

- idx_movies_title on **Movies(title)** for quicker movie title lookups.

Eliminates full table scans when looking up a movie by name (WHERE title = %s).
It speeds up query_6, which searches by title to find the relevant movie (first CTE).

Note that indexes on foreign keys are created automatically by mysql to enforce FK constraints in tables on insertions.
They help speed up joins on FKs, e.g. in query_5, join of Production_Companies and Movies_Production_Companies on production_company_id that is defined as a FK in Movies_Production_Companies. The column production_company_id is indexed in Production_Companies since it's a PK and indexed in the second table, since it's a FK there. Thus, the join is optimized.

4. Main Queries and Database Support

The system supports six main queries, optimized for performance and usability.

Additional considerations; query optimizations. A well-designed schema must be complemented by optimized queries to ensure that the system performs efficiently. This includes the use of query execution optimizations, and advanced SQL techniques such as Common Table Expressions (CTEs), window functions, and efficient joins. These query optimizations are as integral to performance as primary keys, foreign keys, normalization and indexes. For example:

- CTEs allow us to precompute aggregates (such as average genre ratings) and avoid redundant recalculations, making complex analytical queries more efficient.
- Using window functions can eliminate the need for costly correlated subqueries, improving performance.
- Properly designed joins on indexed columns prevent unnecessary full table scans, ensuring queries remain fast even as data grows.

Query 1: Search for Movies by Overview Keywords

- **Purpose**: Finds movies whose overviews contain specified keyword(s), ranking them by relevance(text match score) and popularity. Returns top 10 results by default, with an adjustable limit up to 10000.
- **Supported by Database Design**:
    - Optimization: Utilizes FULLTEXT indexing on the Movies.overview column for fast, scalable keyword-based retrieval.
    - No Joins: All movie metadata is stored in the Movies table, making the query self-contained. No joins are required , which significantly improves performance.

Query 2: Find Actor Dynasties by Last Name

- **Purpose**: Identifies actor dynasties (families in the film industry) by searching for actors with the same last name. Can also find actors with the same first name, though this is less meaningful. Ranks actors by the number of movies they have appeared in.
- **Supported by Database Design**:
    - Optimization: Utilizes FULLTEXT indexing on Actors.name column for fast, scalable keyword-based retrieval.
    - Efficient Joins on Indexed Columns: Actors.actor_id, Movies_Actors.actor_id are indexed as PK and FK respectively, allowing efficient joins between actors and movies.

Query 3: Find Top 5 Most Profitable Movies in a Genre

- **Purpose**: Identifies the top 5 most profitable movies in a given genre. Compares each movie's profit against the genre's average profit.
- **Supported by Database Design**:
    - Optimization: Uses B-tree indexing on (revenue, budget) for efficient profit calculation.
    - Efficient Joins on Indexed Columns: movie_id, genre_id are PKs/FKs in their relative tables and are indexed. Joins of Movies, Movies_Genres, and Genres are performed efficiently.
- **Query Optimizations**:
    - Window Function (`AVG() OVER (PARTITION BY g.genre_name)`): Computes genre-wide average profit without recalculating for each row. Faster than a correlated subquery that recalculates AVG() multiple times.

Query 4: Find Top 10 High-Rated Actors in a Genre

- **Purpose**: Identifies actors who have appeared in movies of a given genre where the movie's rating is higher than the average rating for that genre. Only movies from the specified genre are considered. Actors are ranked by the number of qualifying high-rated movies they have appeared in.
- **Supported by Database Design**:
    - Optimization:
        - B-tree Index on Genres.genre_name :
          Speeds up filtering by genre (`WHERE g.genre_name = %s`).
        - B-tree Index on Movies.vote_average :
          Speeds up filtering for movies with above-average ratings.
    - Efficient Joins on Indexed Columns: Joins Actors, Movies_Actors, Movies, Movies_Genres, and Genres using indexed primary and foreign keys (to filter high-rated movies of the given genre and count the number of such movies for each actor).
- **Query Optimizations**:
    - CTE for Efficient Computation: precomputes the genre's average rating once, avoiding redundant recalculations.

Query 5: Find Top 5 Production Companies by Revenue

- **Purpose**: Identifies top 5 most successful production companies ranked by total revenue, considering only larger companies that have produced more than 5 movies.
  Tiebreaker: average revenue per movie.
- **Supported by Database Design**:
    - Optimization:
        - B-tree Index on Movies.revenue (Covered by Composite Index idx_movies_revenue_budget) :
          Optimizes queries that aggregate Movies.revenue (e.g., SUM(revenue), AVG(revenue)). Prevents full table scans by allowing mysql to scan the index instead of the entire Movies table.
        - B-tree index on Movies_Production_Companies.production_company_id :
          (Automatically created for FK). Speeds up joins with Production_Companies.

Query 6: Suggest Related Movies Based on Shared Keywords

- **Purpose**: Finds movies related to a given title based on shared keywords.
  Ranks results by the number of shared keywords (higher matches come first).
  If multiple movies have the given title, selects the most popular one.
  Returns up to 10 suggestions by default (configurable up to 10,000).
- **Supported by Database Design**:
    - Optimization:
      B-tree Index on Movies.title. Speeds up movie lookup (`WHERE title = %s.`
    - Efficient Joins on Indexed Columns: Joins Movies_Keywords, Movies on indexed FK and PK movie_id, joins MovieKeywords (the CTE) with Movies_Keywords on indexed Movies_Keywords.keyword_id FK.

- **Query Optimizations**:
    - CTE for Efficient Computation:
        - CTE MovieID finds the most popular version of the input movie title.
        - CTE MovieKeywords retrieves all keyword IDs associated with the selected movie.
- **Efficient Filtering and Ranking**
    - Excludes the original movie using `WHERE mk.movie_id NOT IN (SELECT movie_id FROM MovieID)`.
    - Ranks results first by number of shared keywords (higher is better), then by popularity as a tiebreaker.

The database schema supports these queries efficiently by leveraging foreign keys, indexing, and optimized query structures.

5. Code Structure and API Usage

This section outlines the project's code structure and the API usage for data gathering.

5.1 Code Structure

The project consists of multiple modules, each handling a specific task related to data retrieval, database management, and query execution.

Configuration

- /config/config.py
  Stores database connection settings (host, port, user, password, database).
  Specifies dataset paths for local storage after retrieval.

API Authentication

- /.kaggle/kaggle.json
  Kaggle API authentication file, containing API credentials (username, key).
  Used to authenticate API requests when downloading datasets from Kaggle.

Database Schema & Data Loading

- /src/create_db_script.py
  Contains code responsible for creating the database.
  Creates the database schema, defining tables
  `Movies, Genres, Movies_Genres, Keywords, Movies_Keywords, Production_Companies, Movies_Production_Companies, Actors, Movies_Actors.`
  Adds indexes (FullText, B-Tree) to improve query performance.
  Supports dropping all tables if needed.
  Supports downloading and extracting a dataset from Kaggle using the Kaggle API.

Data Retrieval

- /src/api_data_retrieve.py
  Handles data insertion.
  Processes downloaded movie data from Kaggle. Handles missing values.
  Transforms JSON fields (genres, keywords, etc.) into structured database entries.
  Executes batch inserts (INSERT INTO ... VALUES (...)) for efficient data insertion via `cursor.executemany()`.

Query Execution

- /src/queries_db_script.py
  Includes functions for the DB queries (query NUM).
  Defines six optimized mysql queries for retrieving valuable data and key findings from the database:
  1. Search for movies by overview keywords (FULLTEXT search).
  2. Find actor dynasties by last name (FULLTEXT search).
  3. Find top 5 most profitable movies in a genre.
  4. Find top 10 high-rated actors in a genre.
  5. Find top 5 production companies by revenue.
  6. Suggest related movies based on shared keywords.
- /src/queries_execution.py
  Includes the main function and provides example usages of the queries from queries_db_script.py.
  execute_query(connection, query_func, *args, max_width=32) executes queries.

main() provides an example of using all 6 main queries. Commented out code can be used to drop all tables and to create schema and populate an empty database.

Documentation

/documentation/

- system_docs.pdf - This file.
- user manual.pdf -
  An overview of the application's functionality, presentation of the designed application's features.
- name_and_id.txt - Team members IDs and names.
- mysql_and_user_password.txt - The mysql user and password assigned for mysql1 server.

requirements.txt Python requirements file.

5.2 API Usage: Gathering Data from Kaggle

This project retrieves movie metadata using the Kaggle API.
The dataset "tmdb/tmdb-movie-metadata" is downloaded from Kaggle.
The API authenticates requests using kaggle.json.
The data is stored in a local directory (data/) and loaded into the database.

The data is retrieved from Kaggle using function `download_and_extract_dataset()`.

Key Steps:

1. Sets up Kaggle API authentication using kaggle.json.
2. Downloads the TMDB dataset (tmdb_5000_movies.csv, tmdb_5000_credits.csv).
3. Extracts and saves CSV files to the local directory.

The data is loaded into the database using function `load_data_to_database()`.

Key Steps:

1. Reads movie metadata from CSV files.
2. Processes JSON fields (e.g., genres, keywords) into structured formats.
3. Inserts processed data into respective database tables (Movies, Genres, etc.) using function `insert_data()` that performs quick batch inserts.