System Documentation

This document provides an overview of the database schema structure for a movies dataset, explains the reasoning for the chosen design (including discussions of efficiency and drawbacks of alternatives), and describes database optimizations such as index usage.

1. Database Schema Structure

1.1 Entity-Relationship Overview

Entity-Relationship Diagram (Conceptual)

```
movies ---< movie_genres >--- genres
  |
  |---< movie_cast >--- people
  |
  |---< movie_crew >--- people
  |
  |---< movie_keywords >--- keywords
  |
  |---< movie_ratings_summary (1-to-1)
```

The schema is divided into the following entities and relationship (junction) tables:

- **movies** – Central table containing metadata about each movie (title, release date, budget, revenue, ratings, etc.).
- **genres** – Lookup table for genre names (e.g., 'Action', 'Comedy', 'Drama').
- **movie_genres** – Junction table linking movies to genres (many-to-many relationship).
- **people** – Lookup table for person information (actors, directors, crew members).
- **movie_cast** – Junction table linking movies to cast members with their roles and cast order.
- **movie_crew** – Junction table linking movies to crew members with their department and job.
- **keywords** – Lookup table for movie keywords/tags.
- **movie_keywords** – Junction table linking movies to keywords (many-to-many relationship).
- **movie_ratings_summary** – Aggregated rating information (average rating and count) per movie.

1.2 Table Definitions

1. movies

```sql
CREATE TABLE movies (
    movie_id            INT PRIMARY KEY,
    title               VARCHAR(255),
    original_title      VARCHAR(255),
    original_language   VARCHAR(10),
    release_date        DATE,
    release_year        INT,
    runtime             INT,
    budget              BIGINT,
    revenue             BIGINT,
    popularity          DECIMAL(10, 6),
    vote_average        DECIMAL(3, 1),
    vote_count          INT,
    tagline             TEXT,
    overview            TEXT,
    INDEX idx_revenue (revenue),
    INDEX idx_vote_average (vote_average),
    FULLTEXT idx_ft_title (title),
    FULLTEXT idx_ft_overview (overview)
);
```

2. genres

```sql
CREATE TABLE genres (
    genre_id    INT PRIMARY KEY,
    name        VARCHAR(100) NOT NULL UNIQUE
);
```

3. movie_genres (junction table)

```sql
CREATE TABLE movie_genres (
    movie_id    INT,
    genre_id    INT,
    PRIMARY KEY (movie_id, genre_id),
    CONSTRAINT fk_movie_genres_movie
        FOREIGN KEY (movie_id)
        REFERENCES movies(movie_id)
        ON DELETE CASCADE
        ON UPDATE CASCADE,
    CONSTRAINT fk_movie_genres_genre
        FOREIGN KEY (genre_id)
        REFERENCES genres(genre_id)
        ON DELETE CASCADE
        ON UPDATE CASCADE
);
```

4. people

```
CREATE TABLE people (
    person_id   INT PRIMARY KEY,
    name        VARCHAR(255) NOT NULL
);
```

5. movie_cast (junction table)

```
CREATE TABLE movie_cast (
    movie_id        INT,
    person_id       INT,
    cast_order      INT,
    character_name  VARCHAR(500),
    PRIMARY KEY (movie_id, person_id, cast_order),
    INDEX idx_movie_cast_order (movie_id, cast_order, person_id),
    CONSTRAINT fk_movie_cast_movie
        FOREIGN KEY (movie_id)
        REFERENCES movies(movie_id)
        ON DELETE CASCADE
        ON UPDATE CASCADE,
    CONSTRAINT fk_movie_cast_person
        FOREIGN KEY (person_id)
        REFERENCES people(person_id)
        ON DELETE CASCADE
        ON UPDATE CASCADE
);
```

6. movie_crew (junction table)

```
CREATE TABLE movie_crew (
    movie_id    INT,
    person_id   INT,
    department  VARCHAR(100),
    job         VARCHAR(255),
    PRIMARY KEY (movie_id, person_id, department, job),
    INDEX idx_job (job),
    CONSTRAINT fk_movie_crew_movie
        FOREIGN KEY (movie_id)
        REFERENCES movies(movie_id)
        ON DELETE CASCADE
        ON UPDATE CASCADE,
    CONSTRAINT fk_movie_crew_person
        FOREIGN KEY (person_id)
        REFERENCES people(person_id)
        ON DELETE CASCADE
```

```
        ON UPDATE CASCADE
);
```

7. keywords

```
CREATE TABLE keywords (
    keyword_id  INT PRIMARY KEY,
    name        VARCHAR(255) NOT NULL
);
```

8. movie_keywords (junction table)

```
CREATE TABLE movie_keywords (
    movie_id    INT,
    keyword_id  INT,
    PRIMARY KEY (movie_id, keyword_id),
    CONSTRAINT fk_movie_keywords_movie
        FOREIGN KEY (movie_id)
        REFERENCES movies(movie_id)
        ON DELETE CASCADE
        ON UPDATE CASCADE,
    CONSTRAINT fk_movie_keywords_keyword
        FOREIGN KEY (keyword_id)
        REFERENCES keywords(keyword_id)
        ON DELETE CASCADE
        ON UPDATE CASCADE
);
```

9. movie_ratings_summary

```
CREATE TABLE movie_ratings_summary (
    movie_id        INT PRIMARY KEY,
    rating_avg      DECIMAL(3, 2),
    rating_count    INT,
    CONSTRAINT fk_movie_ratings_summary_movie
        FOREIGN KEY (movie_id)
        REFERENCES movies(movie_id)
        ON DELETE CASCADE
        ON UPDATE CASCADE
);
```

1.3 Summary of Primary and Foreign Keys

1.3.1 Primary Keys

- **movies**: movie_id
- **genres**: genre_id

- **people**: person_id
- **keywords**: keyword_id
- **movie_genres** (junction table): Composite key of (movie_id, genre_id)
- **movie_cast** (junction table): Composite key of (movie_id, person_id, cast_order)
- **movie_crew** (junction table): Composite key of (movie_id, person_id, department, job)
- **movie_keywords** (junction table): Composite key of (movie_id, keyword_id)
- **movie_ratings_summary**: movie_id (also serves as foreign key)

1.3.2 Foreign Keys

- **movie_genres**:
  - movie_id → movies(movie_id)
  - genre_id → genres(genre_id)
- **movie_cast**:
  - movie_id → movies(movie_id)
  - person_id → people(person_id)
- **movie_crew**:
  - movie_id → movies(movie_id)
  - person_id → people(person_id)
- **movie_keywords**:
  - movie_id → movies(movie_id)
  - keyword_id → keywords(keyword_id)
- **movie_ratings_summary**:
  - movie_id → movies(movie_id)

2. Design Reasoning and Alternatives

2.1 Normalization vs. JSON Storage

Originally, fields like genres, cast, crew, and keywords were stored as JSON arrays in the source data. While JSON storage can be convenient for unstructured data, it makes relational queries inefficient.

By **normalizing** these fields into separate tables, we:

- Achieve cleaner relationships (1-to-M or M-to-M) using foreign keys.
- Facilitate more flexible querying (e.g., joins, filtering by genre, finding actor pairs).
- Ensure data integrity and avoid duplication of information.
- Enable efficient indexing for performance optimization.

An alternative design could keep the JSON columns in movies for simplicity, but it would lead to: - Slower queries involving array data (requires parsing JSON in queries) - No referential integrity (cannot enforce foreign key constraints) -

More complex application-level parsing - Difficulty in indexing nested JSON data

2.2 Why Separate Lookup Tables?

- **genres**, **people**, and **keywords** serve as reference data (lookup tables) with their own natural primary keys.
- Using separate lookup tables:
  - Keeps the dataset more structured
  - Ensures consistent usage of IDs/codes
  - Makes it easier to track or update one entity (e.g., correcting a person's name) in one place
  - Prevents data duplication and inconsistencies
  - Enables efficient indexing on lookup tables

2.3 One-to-Many and Many-to-Many Decisions

- **Many-to-Many Relationships**: A movie can have multiple genres, keywords, cast members, and crew members. This requires junction tables:
  - **movie_genres**: Links movies to multiple genres
  - **movie_keywords**: Links movies to multiple keywords
  - **movie_cast**: Links movies to multiple people (with additional attributes: cast_order, character_name)
  - **movie_crew**: Links movies to multiple people (with additional attributes: department, job)
- **One-to-One Relationship**: Each movie has one aggregated rating summary, so **movie_ratings_summary** uses movie_id as both primary key and foreign key.

2.4 Composite Primary Keys in Junction Tables

Junction tables use composite primary keys to: - Ensure uniqueness of relationships (e.g., a person can only appear once in a movie's cast with a given cast_order) - Automatically create indexes on the composite key for efficient joins - Prevent duplicate entries

Special cases: - **movie_cast**: Includes cast_order in the primary key to allow the same person to appear multiple times in different roles (though rare) - **movie_crew**: Includes department and job to allow the same person to have multiple roles in the same movie

2.5 Drawbacks of Alternatives

- **All-in-one table**: Storing all attributes in a single table (including repeated or multi-valued attributes) leads to high redundancy and violates normalization principles.

- **Repeated columns**: A different approach could add repeated columns (e.g., genre1, genre2, genre3, actor1, actor2, etc.), but this will:
  - Limit the number of possible attributes
  - Require schema changes for new entries
  - Make queries more complex (multiple OR conditions)
  - Waste storage space
- **JSON fields only**: While flexible, they do not provide:
  - Robust relational integrity
  - Straightforward indexing for typical queries
  - Efficient joins and aggregations
  - Easy enforcement of data constraints

3. Database Optimizations

3.1 Index Usage

3.1.1 Primary Keys Every table has a primary key, which automatically creates a clustered index in MySQL (InnoDB). This ensures: - Fast lookups by primary key - Uniqueness constraints - Efficient foreign key references

3.1.2 Foreign Keys Each junction table has foreign keys referencing the parent tables. MySQL automatically creates indexes on foreign key columns, which: - Ensures referential integrity - Optimizes joins between related tables - Speeds up cascading operations (ON DELETE CASCADE, ON UPDATE CASCADE)

3.1.3 Additional Indexes movies table: - `INDEX idx_revenue (revenue)`: Optimizes queries that filter or sort by revenue (Query 1, Query 4, Query 5) - `INDEX idx_vote_average (vote_average)`: Optimizes sorting by rating in Query 3 - `FULLTEXT idx_ft_title (title)`: Enables full-text search on movie titles (Query 2) - `FULLTEXT idx_ft_overview (overview)`: Enables full-text search on movie overviews/plots (Query 1)

movie_cast table: - `INDEX idx_movie_cast_order (movie_id, cast_order, person_id)`: Optimizes Query 3 by: - Allowing efficient filtering of cast_order < 10 - Supporting the self-join on movie_id - Including person_id for the comparison condition

movie_crew table: - `INDEX idx_job (job)`: Optimizes Query 4 by quickly filtering for job = 'Director'

3.1.4 Composite Primary Key Indexes The composite primary keys in junction tables also serve as indexes: - **movie_genres (movie_id, genre_id)**: Enables efficient lookups of all genres for a movie (and vice versa) - used in Query 5 - **movie_cast (movie_id, person_id, cast_order)**:

Supports Query 3's self-join operations - **movie_keywords (movie_id, keyword_id)**: Enables efficient keyword lookups

3.2 Index Selection Rationale

Indexes were chosen based on query patterns:

1. **Revenue indexing**: Many queries filter or sort by revenue, making `idx_revenue` essential for performance.
2. **Full-text indexes**: Required for Query 1 and Query 2, which perform text searches. FULLTEXT indexes use MySQL's specialized full-text search engine.
3. **Composite indexes**: Query 3's self-join benefits from `idx_movie_cast_order`, which allows filtering and joining in a single index scan.
4. **Job indexing**: Query 4 filters by job = 'Director', making `idx_job` necessary for efficient filtering.

4. Customized Queries

This section provides an overview and explanation of the five main queries, their purpose, and how the database design supports them.

4.1 Query 1: Plot/Concept Analysis (Full-text Search)

Purpose: Allows producers to search for plot keywords (e.g., "apocalypse", "wedding") to see the financial performance of similar past movies.

Query:

```
SELECT
    title,
    release_year,
    CONCAT('$', FORMAT(budget, 0)) as budget_formatted,
    CONCAT('$', FORMAT(revenue, 0)) as revenue_formatted,
    ROUND(revenue / NULLIF(budget, 0), 2) as roi_ratio
FROM movies
WHERE
    MATCH(overview) AGAINST (%s IN NATURAL LANGUAGE MODE)
    AND budget > 0
    AND revenue > 0
ORDER BY revenue DESC
LIMIT 20;
```

Explanation: - Uses MySQL's full-text search on the `overview` column to find movies with similar plots/concepts - Filters to movies with valid budget and revenue data - Calculates ROI ratio (revenue/budget) - Orders by revenue to show most financially successful matches - Formats budget and revenue as currency strings for readability

Database Design Support: - The `FULLTEXT idx_ft_overview` index enables fast full-text searching - The `idx_revenue` index optimizes the ORDER BY revenue operation - Normalized design allows efficient filtering and sorting

4.2 Query 2: Title Competitor Check (Full-text Search)

Purpose: Search for specific phrases in titles to analyze popularity and viewer reception of similar-titled movies.

Query:

```
SELECT
    title,
    popularity,
    vote_average,
    vote_count
FROM movies
WHERE MATCH(title) AGAINST (%s IN NATURAL LANGUAGE MODE)
ORDER BY popularity DESC
LIMIT 20;
```

Explanation: - Uses full-text search on the `title` column to find movies with similar titles - Returns popularity metrics and ratings to help assess competitor performance - Orders by popularity to show the most popular matches

Database Design Support: - The `FULLTEXT idx_ft_title` index enables fast title searching - Normalized structure allows quick access to popularity and rating data

4.3 Query 3: Best Actor Combinations (Pairs) by Rating (Complex Query)

Purpose: Identifies which actor pairs ("power couples") have worked best together, ranked by average rating of their collaborations.

Query:

```
SELECT
    p1.name AS actor_1,
    p2.name AS actor_2,
    COUNT(*) AS movies_together,
    ROUND(AVG(m.vote_average), 2) AS avg_rating
FROM (
    SELECT movie_id, person_id, cast_order
    FROM movie_cast
    WHERE cast_order < 10
) mc1
JOIN (
    SELECT movie_id, person_id, cast_order
```

```
    FROM movie_cast
    WHERE cast_order < 10
) mc2 ON mc1.movie_id = mc2.movie_id AND mc1.person_id < mc2.person_id
JOIN people p1 ON mc1.person_id = p1.person_id
JOIN people p2 ON mc2.person_id = p2.person_id
JOIN movies m ON mc1.movie_id = m.movie_id
GROUP BY p1.person_id, p2.person_id, p1.name, p2.name
HAVING COUNT(*) >= %s
ORDER BY avg_rating DESC
LIMIT 15;
```

Explanation: - Performs a self-join on `movie_cast` to find all pairs of actors who appeared in the same movie - Filters to main actors only (cast_order < 10) to focus on significant collaborations - Uses `mc1.person_id < mc2.person_id` to avoid duplicate pairs (A,B) and (B,A) - Groups by actor pairs and calculates average rating across their collaborations - Filters to pairs who worked together at least N times (HAVING clause) for statistical significance - Orders by average rating to show the best-performing pairs

Database Design Support: - The normalized `movie_cast` table with composite primary key enables efficient self-joins - The `idx_movie_cast_order` index optimizes filtering and joining operations - The `idx_vote_average` index on movies table optimizes sorting by rating - Separate `people` table allows efficient name lookups

Complexity Elements: - Self-join (joining a table to itself) - GROUP BY with aggregation (COUNT, AVG) - HAVING clause (filtering on aggregated data) - Subqueries for early filtering

4.4 Query 4: Best Directors by Revenue (Complex Query)

Purpose: Identifies the most commercially successful directors ranked by total revenue across all their movies.

Query:

```
SELECT
    p.name AS director_name,
    COUNT(m.movie_id) AS movies_directed,
    CONCAT('$', FORMAT(SUM(m.revenue), 0)) AS total_revenue
FROM people p
JOIN movie_crew mc ON p.person_id = mc.person_id
JOIN movies m ON mc.movie_id = m.movie_id
WHERE mc.job = 'Director' AND m.revenue > 0
GROUP BY p.person_id, p.name
ORDER BY SUM(m.revenue) DESC
LIMIT %s;
```

Explanation: - Joins `people`, `movie_crew`, and `movies` tables to link directors

with their movies - Filters for job = 'Director' and movies with revenue > 0 - Groups by director and aggregates: - Counts number of movies directed - Sums total revenue across all movies - Orders by total revenue to show most commercially successful directors - Formats revenue as currency string

Database Design Support: - The normalized `movie_crew` table with department and job fields enables filtering by role - The `idx_job` index optimizes filtering for job = 'Director' - The `idx_revenue` index on movies table optimizes sorting - Foreign key indexes enable efficient joins between tables

Complexity Elements: - JOIN operations across multiple tables - GROUP BY with aggregation (COUNT, SUM) - ORDER BY on aggregated data

4.5 Query 5: Best Genre Combinations by Revenue (Complex Query)

Purpose: Helps producers decide on genre mashups (e.g., "Action-Comedy" vs "Horror-Romance") by showing which genre pairs perform best financially.

Query:

```
SELECT
    g1.name AS genre_1,
    g2.name AS genre_2,
    COUNT(m.movie_id) AS movie_count,
    CONCAT('$', FORMAT(AVG(m.revenue), 0)) AS avg_revenue
FROM movie_genres mg1
JOIN movie_genres mg2 ON mg1.movie_id = mg2.movie_id
JOIN genres g1 ON mg1.genre_id = g1.genre_id
JOIN genres g2 ON mg2.genre_id = g2.genre_id
JOIN movies m ON mg1.movie_id = m.movie_id
WHERE mg1.genre_id < mg2.genre_id
  AND m.revenue >= %s
GROUP BY g1.genre_id, g2.genre_id, g1.name, g2.name
ORDER BY AVG(m.revenue) DESC
LIMIT 15;
```

Explanation: - Performs a self-join on `movie_genres` to find all genre pairs that appear together in movies - Uses `mg1.genre_id < mg2.genre_id` to avoid duplicate pairs (Action-Comedy vs Comedy-Action) - Filters to movies with revenue above a threshold - Groups by genre pairs and calculates: - Count of movies with that genre combination - Average revenue for that combination - Orders by average revenue to show most profitable genre combinations - Formats revenue as currency string

Database Design Support: - The normalized `movie_genres` junction table with composite primary key enables efficient self-joins - The composite primary key (movie_id, genre_id) serves as an index for the join operation - The `idx_revenue` index on movies table optimizes filtering and sorting - Separate `genres` table allows efficient name lookups

Complexity Elements: - Self-join (joining a table to itself) - GROUP BY with aggregation (COUNT, AVG) - ORDER BY on aggregated data - Multiple JOIN operations

5. Code Structure and API Usage

5.1 Project Structure

The project follows the required structure:

```
src/
   create_db_script.py      # Database schema creation
   api_data_retrieve.py     # Data insertion from CSV files
   queries_db_script.py     # Query functions (query_1 through query_5)
   queries_execution.py     # Example query executions
```

5.2 Code Organization

create_db_script.py: - Contains functions to create each table - Uses `create_all_tables()` as the main entry point - Creates tables in dependency order (base tables first, then junction tables)

api_data_retrieve.py: - Contains functions to populate tables from CSV files - Uses batch inserts (1000 rows at a time) to handle large datasets efficiently - Handles NULL values and date conversions appropriately - Filters data to respect foreign key constraints (e.g., movie_ratings_summary only includes movies that exist)

queries_db_script.py: - Contains five query functions: `query_1` through `query_5` - Each function accepts user input parameters and returns query results - Functions are well-documented with docstrings explaining purpose and parameters - Uses parameterized queries to prevent SQL injection

queries_execution.py: - Provides example usage of all queries - Demonstrates how to call each query function with sample parameters - Shows proper connection handling and error management

5.3 Configuration

Database credentials are stored in `config.py` in the root directory, allowing easy configuration without modifying source code. All scripts import and use this configuration.

5.4 Data Population Process

1. The relevant CSV files are generated from raw CSV data downloaded from https://www.kaggle.com/datasets/rounakbanik/the-movies-dataset using `table_creator.py` (archived)
2. CSV files are stored in `src/dataSets/`

3. `api_data_retrieve.py` reads CSV files and populates the database
4. Data is inserted in batches to handle large datasets efficiently
5. Foreign key constraints ensure data integrity

5.5 Error Handling

All scripts include appropriate error handling: - Database connection errors are caught and reported - Transaction rollback on errors to maintain data integrity - Clear error messages for debugging

5.6 Dependencies

The project requires: - `mysql-connector-python`: For MySQL database connectivity - `pandas`: For CSV file reading and data processing - `numpy`: For handling NaN values (used by pandas)

All dependencies are listed in `requirements.txt`.