# Database Schema Manual

324827328, 213713522

## Contents

# 1 Introduction

The database design was a critical component of the project, establishing a foundation for API performance and application integrity. We aimed for a normalized schema to minimize redundancy while maintaining the flexibility required for analytical queries.

# 2 The Core Entities

## 2.1 Movies

The `movies` table is the central entity of the database. We explicitly included `vote_average` and `vote_count` fields in this table.

This denormalization strategy was chosen to optimize read performance. Instead of calculating averages from the `ratings` table for every query, we maintain these aggregated values transactionally.

```sql
CREATE TABLE IF NOT EXISTS movies (
    id INT PRIMARY KEY,
    title VARCHAR(500) NOT NULL,
    -- We keep the aggregated stats right here for speed
    vote_average FLOAT,
    vote_count INT,
    -- ... other metadata
    budget BIGINT,
    revenue BIGINT,
    status VARCHAR(50)
);
```

Listing 1: The heart of the system

## 2.2 People

We unified actors, directors, and writers into a single `people` table. This approach avoids data duplication for individuals who fulfill multiple roles (e.g., both acting and directing).

```sql
CREATE TABLE IF NOT EXISTS people (
    id INT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    gender INT,
    profile_path VARCHAR(255)
);
```

Listing 2: A unified People table

# 3 Connecting the Dots: Relationships

The database relational structure defines the connections between entities.

## 3.1 The Cast & Crew

Connecting `movies` to `people` required more than a simple link. We needed to know *what role* they played. For the **Cast**, we stored the `character_name` and an `order_index`. The index is crucial because it allows the frontend to validly display the "Top Billed" actors first, rather than a random alphabetical list. For the **Crew**, it's about the `job` and `department`. This granular detail allows us to answer interesting questions like "Which directors also acted in their own movies?"

```sql
CREATE TABLE IF NOT EXISTS cast_members (
    credit_id VARCHAR(255) PRIMARY KEY,
    movie_id INT,
    person_id INT,
    character_name VARCHAR(500),
    order_index INT,
    FOREIGN KEY (movie_id) REFERENCES movies(id),
    FOREIGN KEY (person_id) REFERENCES people(id)
);
```

Listing 3: Linking People to Movies

### 3.2 Categorization (Genres, Keywords)

We handled categories like Genres and Keywords as Many-to-Many relationships. This means a movie can be both "Action" and "Comedy". We separated the names into their own lookup tables (`genres`, `keywords`) and used junction tables (`movie_genres`) to link them. This makes searching for "All Action movies" incredibly fast because it's just a simple index lookup.

## 4 User Interaction

### 4.1 Ratings

We built the `ratings` table. We decided to use a string-based `user_id` to align with our "Soft Session" strategy on the client. This table grows the fastest, so we kept it lightweight: just who, what, what score, and when.

```
CREATE TABLE IF NOT EXISTS ratings (
    id INT AUTO_INCREMENT PRIMARY KEY,
    movie_id INT,
    user_id VARCHAR(255) NOT NULL,
    rating FLOAT,
    timestamp DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (movie_id) REFERENCES movies(id)
);
```

Listing 4: Capturing user votes

## 5 Binary Storage

### 5.1 Storing Images in the DB

We opted to store movie posters directly in the database using the `LONGBLOB` data type. While external storage (like S3) is a common pattern, storing images within the database ensures the dataset is self-contained. This simplifies backup and deployment procedures by keeping the entire application state in a single location.

```
CREATE TABLE IF NOT EXISTS movie_posters (
    movie_id INT PRIMARY KEY,
    image LONGBLOB,
    FOREIGN KEY (movie_id) REFERENCES movies(id)
);
```

Listing 5: Self-contained image storage