Database Schema Manual

324827328, 213713522

Contents

1 Introduction

Designing the database was the foundation of this entire project. We knew that if we got the data model wrong, everything else—from the API performance to the client UI—would suffer. Our goal was to create a schema that was strictly normalized to prevent data redundancy, yet flexible enough to answer complex analytical questions. We used MySQL 8.4 because of its reliability and its strict enforcement of referential integrity.

2 The Core Entities

2.1 Movies

At the heart of the system is the `movies` table. This isn't just a list of titles; it's a carefully chosen set of attributes that defines what a "movie" is in our application. We decided to store the `vote_average` and `vote_count` directly on this table.

This was a deliberate trade-off. In a perfectly normalized world, you would calculate the average from the `ratings` table every time. But we realized that calculating an average from thousands of rows every time a user loads the home page would be too slow. So, we utilized a "denormalization" strategy where we update these fields transactionally.

```
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 treated people—actors, directors, writers—as a single entity. It didn't make sense to have separate tables for "Actors" and "Directors" because often the same person (like Clint Eastwood) does both. By having a unified `people` table, we avoided duplicating data for multi-talented individuals.

```
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 real power of our database comes from how these entities connect.

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 to capture the user voice. 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.

```sql
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 made a controversial choice: storing the movie posters directly in the database using the `LONGBLOB` type. Typically, you would store images on a file server (like AWS S3) and just keep the URL in the database. However, for this project, we wanted the database to be completely self-contained. If you back up the database, you back up the entire application state—images and all. It increased the storage size, but it drastically simplified deployment and backup management.

```sql
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