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.

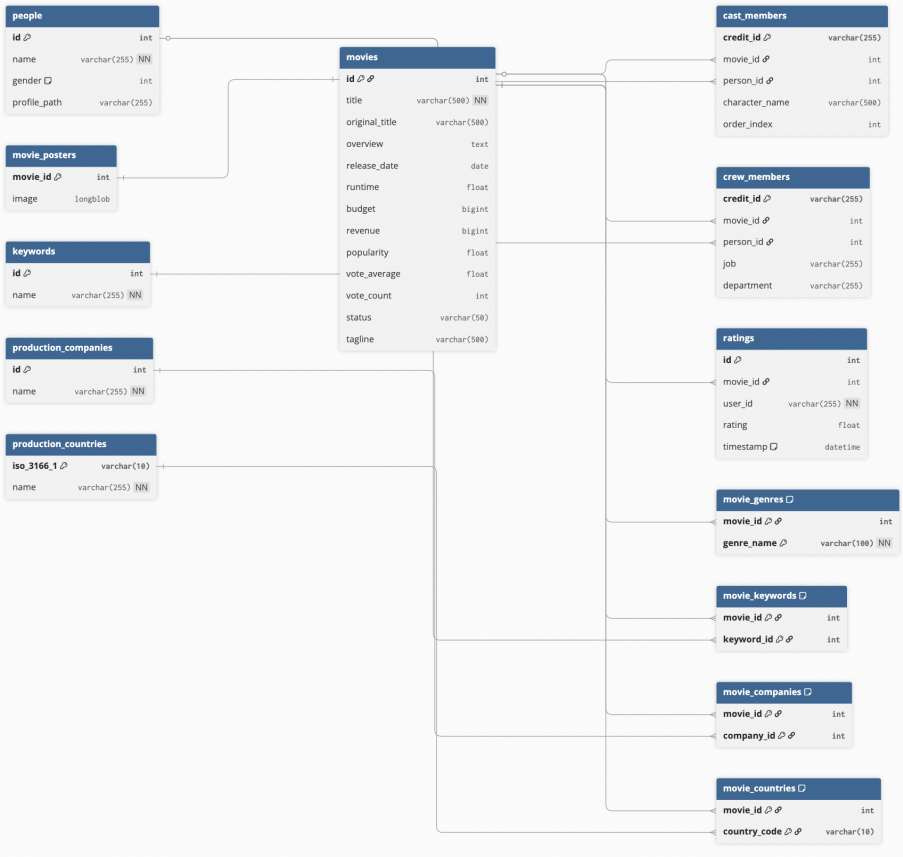

Figure 1: Full Database Schema Diagram

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).

```
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 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?"

```
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.

```
CREATE TABLE IF NOT EXISTS movie_genres (
    movie_id INT,
    genre_name VARCHAR(100) NOT NULL,
    PRIMARY KEY (movie_id, genre_name),
    FOREIGN KEY (movie_id) REFERENCES movies(id)
);
```

Listing 4: Genre Association

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

Listing 5: Keywords Definition

```
CREATE TABLE IF NOT EXISTS movie_keywords (
    movie_id INT,
    keyword_id INT,
    PRIMARY KEY (movie_id, keyword_id),
    FOREIGN KEY (movie_id) REFERENCES movies(id),
    FOREIGN KEY (keyword_id) REFERENCES keywords(id)
);
```

Listing 6: Keyword Association

4 Production Details

To provide a comprehensive view of the film's origin, we track both the production companies and the countries involved. These are modeled as many-to-many relationships, acknowledging that modern films are often co-productions.

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

CREATE TABLE IF NOT EXISTS movie_companies (
    movie_id INT,
    company_id INT,
    PRIMARY KEY (movie_id, company_id),
    FOREIGN KEY (movie_id) REFERENCES movies(id),
    FOREIGN KEY (company_id) REFERENCES production_companies(id)
);
```

Listing 7: Production Companies

```sql
CREATE TABLE IF NOT EXISTS production_countries (
    iso_3166_1 VARCHAR(10) PRIMARY KEY,
    name VARCHAR(255) NOT NULL
);

CREATE TABLE IF NOT EXISTS movie_countries (
    movie_id INT,
    country_code VARCHAR(10),
    PRIMARY KEY (movie_id, country_code),
    FOREIGN KEY (movie_id) REFERENCES movies(id),
    FOREIGN KEY (country_code) REFERENCES production_countries(iso_3166_1)
);
```

Listing 8: Production Countries

5 User Interaction

5.1 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.

```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 9: Capturing user votes

6 Binary Storage

6.1 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.

```sql
CREATE TABLE IF NOT EXISTS movie_posters (
    movie_id INT PRIMARY KEY,
    image LONGBLOB,
```

```
    FOREIGN KEY (movie_id) REFERENCES movies(id)
);
```

Listing 10: Self-contained image storage

7 Application Logic Queries

This section documents the SQL queries used within the application's backend logic.

7.1 Movie Retrieval

For the main listing, we fetch movies with pagination logic.

```
SELECT * FROM movies LIMIT %s OFFSET %s
```

7.2 Movie Details Enrichment

To efficiently load related data for a list of movies (or a single movie), we use `IN` clauses or direct equality checks to fetch Genres, Cast, and Crew.

```
SELECT mg.movie_id, mg.genre_name
FROM movie_genres mg
WHERE mg.movie_id IN ({placeholders})
```

Listing 11: Fetching Genres

```
SELECT c.movie_id, p.id, p.name, p.gender, p.profile_path,
       c.character_name, c.order_index
FROM people p
JOIN cast_members c ON p.id = c.person_id
WHERE c.movie_id IN ({placeholders})
ORDER BY c.movie_id, c.order_index
```

Listing 12: Fetching Cast

```
SELECT c.movie_id, p.id, p.name, p.gender, p.profile_path,
       c.job, c.department
FROM people p
JOIN crew_members c ON p.id = c.person_id
WHERE c.movie_id IN ({placeholders})
```

Listing 13: Fetching Crew

7.3 Search

We implement a simple but effective text search using `LIKE` on the movie title.

```
SELECT * FROM movies WHERE title LIKE %s
```

7.4 Rating System

The rating system involves a transactional flow to ensure data consistency.

First, we check if the user has already rated the movie:

```
SELECT rating FROM ratings
WHERE movie_id = %s AND user_id = %s
```

Then, we perform an "Upsert" operation. If the record exists, we update the timestamp and rating; otherwise, we insert a new record.

```
INSERT INTO ratings (movie_id, user_id, rating)
VALUES (%s, %s, %s)
ON DUPLICATE KEY UPDATE
    rating = VALUES(rating),
    timestamp = CURRENT_TIMESTAMP
```

Finally, to keep the read-heavy `movies` table fast, we recalculate and update the aggregated statistics on the movie record itself.

```
UPDATE movies
SET vote_average = %s, vote_count = %s
WHERE id = %s
```

7.5 Assets

Retrieving the binary image data for a movie poster.

```
SELECT image FROM movie_posters WHERE movie_id = %s
```

8 Analytics & Reporting

The system enables complex analytical queries to derive insights from the movie database. We create 10 different queries to analyze the data. This section details the SQL formulation for these reports.

8.1 Genre Popularity

To determine which genres are most prevalent in the database, we aggregate the count of movies for each genre and order them descending.

```
SELECT genre_name AS genre, COUNT(movie_id) AS movie_count
FROM movie_genres
GROUP BY genre_name
ORDER BY movie_count DESC
LIMIT 5;
```

8.2 Actor Reuse (Prolific Actors)

This query identifies actors who have appeared in more than one movie. It uses a `HAVING` clause to filter groups (actors) after aggregation.

```
SELECT p.name as actor_name, COUNT(c.movie_id) as movie_count
FROM people p
JOIN cast_members c ON p.id = c.person_id
GROUP BY p.id, p.name
HAVING movie_count > 1
ORDER BY movie_count DESC
LIMIT 10
```

8.3 Runtime Analysis

We calculate the average runtime for each genre. Since a movie can have multiple genres, it contributes to the average of each genre it belongs to.

```
SELECT mg.genre_name AS genre, AVG(m.runtime) AS avg_runtime
FROM movie_genres mg
JOIN movies m ON mg.movie_id = m.id
WHERE m.runtime IS NOT NULL
GROUP BY mg.genre_name
ORDER BY avg_runtime DESC
LIMIT 10
```

8.4 Financial Underperformance ("Flops")

We identify movies with a significant budget (over $1M) that failed to recover their costs. The loss is calculated directly in the projection as `budget - revenue`.

```
SELECT title, budget, revenue, (budget - revenue) as loss
FROM movies
WHERE budget > 1000000 AND revenue < budget AND revenue > 0
ORDER BY loss DESC
LIMIT 10
```

8.5 Director-Actors

This complex join finds individuals who have both directed and acted in the *same* movie. It requires joining `people` to `cast_members` and `crew_members`, and ensuring the `movie_id` matches across both roles.

```
SELECT p.name, m.title
FROM people p
JOIN cast_members c ON p.id = c.person_id
JOIN crew_members cr ON p.id = cr.person_id AND c.movie_id = cr.movie_id
JOIN movies m ON c.movie_id = m.id
WHERE cr.job = 'Director'
```

8.6 Global Reach

We aggregate production countries to see which nations produce the most content in our database.

```
SELECT pc.name, COUNT(mc.movie_id) as movie_count
FROM production_countries pc
JOIN movie_countries mc ON pc.iso_3166_1 = mc.country_code
GROUP BY pc.name
ORDER BY movie_count DESC
LIMIT 10
```

8.7 Content Richness

We rank movies by the number of associated keywords, using this as a proxy for the depth of metadata available.

```
SELECT m.title, COUNT(mk.keyword_id) as keyword_count
FROM movies m
JOIN movie_keywords mk ON m.id = mk.movie_id
GROUP BY m.id, m.title
ORDER BY keyword_count DESC
LIMIT 10
```

8.8 Genre Complexity

We identify movies that are categorized into more than 3 genres. This helps find films with complex or cross-cutting themes.

```
SELECT m.title, COUNT(mg.genre_name) as genre_count
FROM movies m
JOIN movie_genres mg ON m.id = mg.movie_id
GROUP BY m.id, m.title
HAVING genre_count > 3
ORDER BY genre_count DESC
```

8.9 Golden Era

We determine the "Best Year" for cinema based on the average user rating. We filter for years with at least 3 movies to avoid statistical outliers from years with very few releases.

```
SELECT YEAR(release_date) as year, AVG(vote_average) as avg_vote, COUNT(*) as
    movie_count
FROM movies
WHERE release_date IS NOT NULL
GROUP BY year
HAVING movie_count >= 3
ORDER BY avg_vote DESC
LIMIT 1
```

8.10 Multiskilled Crew

We find crew members who are talented enough to work in multiple different departments (e.g., Sound and Editing). We use `COUNT(DISTINCT cr.department)` to ensure we are counting different functional areas, not just multiple credits in the same department.

```
SELECT p.name, COUNT(DISTINCT cr.department) as dept_count
FROM people p
JOIN crew_members cr ON p.id = cr.person_id
GROUP BY p.id, p.name
HAVING dept_count > 1
ORDER BY dept_count DESC
LIMIT 10
```