SQL : IMDB Movie Database

Your Mission

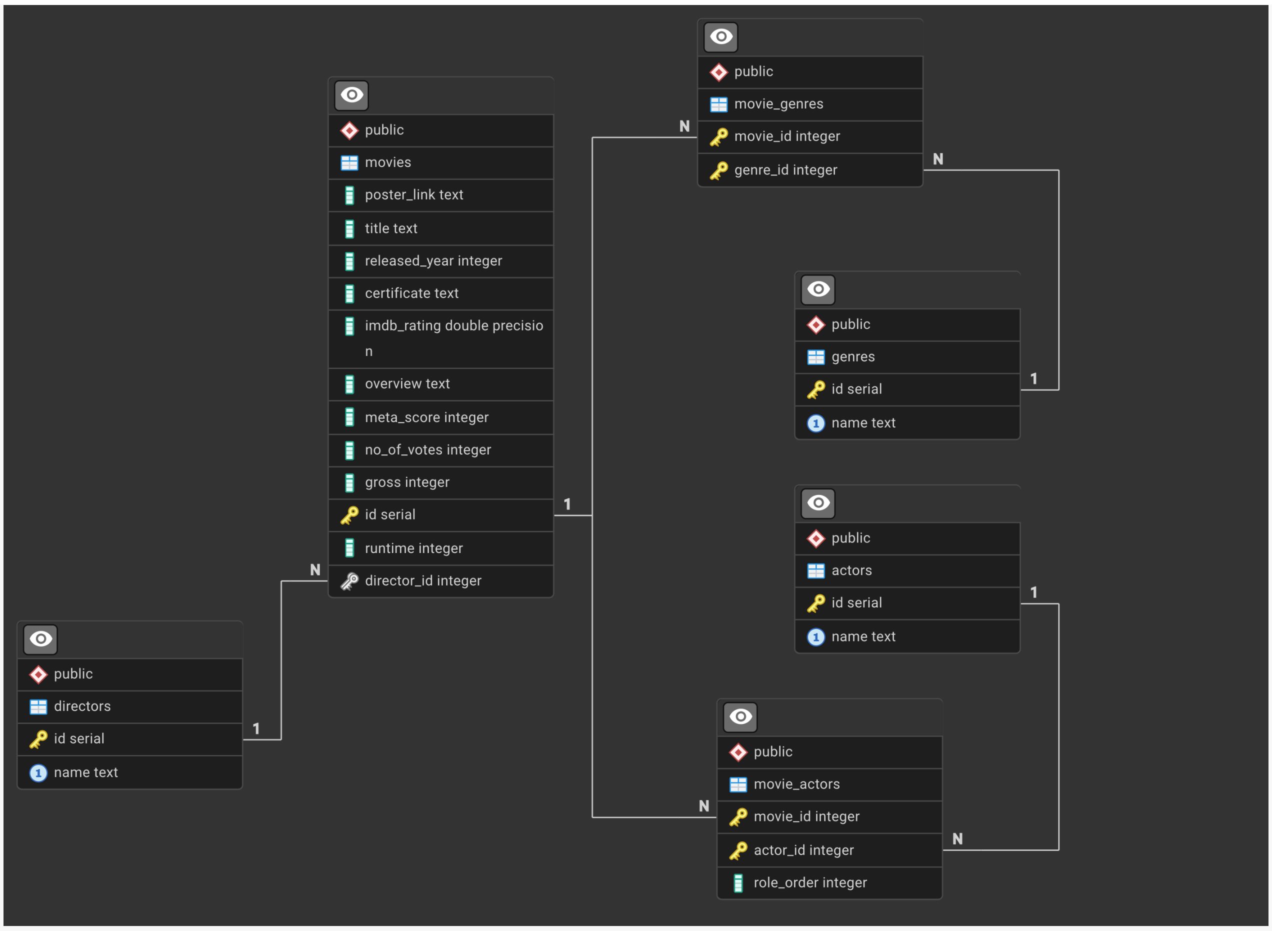
You've just been hired as a Junior Data Analyst at **CineStream**, a new streaming service that wants to make data-driven decisions about which movies to license for their platform.

Your manager has given you access to a database containing information about the top-rated movies from IMDB. She needs you to explore this database and answer several business questions to help the content acquisition team make informed decisions about: - Which genres are most popular among highly-rated films - Which directors and actors appear most frequently in successful movies - What patterns exist in movie ratings and viewer engagement

This is your first day working with SQL (Structured Query Language), the standard language for interacting with databases. Don't worry - we'll guide you through each step!

The IMDB Database Tables

The moviesdb database contains **6 tables** that work together to store movie information:



Main Tables (Storing Core Data)

1. `movies` - The heart of the database. (1000 movies)
    - Contains: Movie titles, release years, ratings, runtime, revenue (gross), and more
    - Each row = One movie
    - Key columns:
        - `id`: Unique identifier for each movie
        - `title`: Movie name
        - `released_year`: When it came out
        - `imdb_rating`: Score from 0-10
        - `no_of_votes`: How many people rated it
        - `runtime`: Length in minutes
        - `director_id`: Links to the directors table
2. `actors` - All actors in the database. (2709 actors)
    - Contains: Actor names
    - Each row = One actor
    - Key columns:
        - `id`: Unique identifier

        - `name`: Actor's full name

3. `directors` - All directors in the database. (548 directors)
    - Contains: Director names
    - Each row = One director
    - Key columns:
        - `id`: Unique identifier
        - `name`: Director's full name
4. `genres` - All possible movie genres. (21 genres)
    - Contains: Genre categories (Action, Drama, Comedy, etc.)
    - Each row = One genre
    - Key columns:
        - `id`: Unique identifier
        - `name`: Genre name

🔗 **Relationship Tables (Connecting Data)**

1. `movie_actors` - Links movies with their actors. (3996 relations)
    - A movie has multiple actors, and actors appear in multiple movies
    - Key columns:
        - `movie_id`: Which movie
        - `actor_id`: Which actor
        - `role_order`: Billing order (1 = lead role, 2 = supporting, etc.)
2. `movie_genres` - Links movies with their genres. (2541 relations)
    - A movie can have multiple genres (e.g., "Action" AND "Comedy"), a genre can be applied to multiple movies
    - Key columns:
        - `movie_id`: Which movie
        - `genre_id`: Which genre

Understanding Table Relationships

One-to-Many Relationship (1:N)

- **movies** ↔ **directors**: Each movie has ONE director, but a director can direct MANY movies
    - The `director_id` in the movies table points to an `id` in the directors table

Many-to-Many Relationships (N:N)

- **movies** ↔ **actors**: A movie has MANY actors, and actors appear in MANY movies

    - Connected through the `movie_actors` table
- **movies** ↔ **genres**: A movie can have MANY genres, and each genre applies to MANY movies
    - Connected through the `movie_genres` table

Load the database

Download the database file (< 1 Mb) from this link

Launch sqlite with the following command: `bash sqlite3 moviesdb.sqlite`

or using a GUI tool like DB Browser for SQLite

Exercises - deliverable

In the following exercises, you will be asked to write SQL queries to answer specific questions about the database. Write your queries in the sqlite session.

Once you are satisfied with the result and copy the SQL query to a file called `queries.sql`. Do not copy the result, just the sql statement.

Submit your queries.sql file at the end of the TP.

SQL Query Exercises

Section A: Your First Queries - Simple SELECT Statements

Let's start by exploring single tables. These queries will help you get comfortable with basic SQL syntax.

A.1 - See All Genres

Task: Display all available genres in the database.

SQL
```sql
-- Your query here:
SELECT * FROM genres;
```

Expected: A list of all genre names with their IDs

A.2 - Find Specific Movie Information

Task: Your manager wants to know the `title`, `release year`, and `IMDB rating` for all movies. Display only these three columns.

```sql
-- Your query here:
SELECT title, released_year, imdb_rating
FROM movies;
```

Hint: Instead of using *, list the specific column names separated by commas

---

A.3 - Browse Recent Movies

Task: Find all movies released after 2015. Show `title`, `release year`, and `IMDB rating`.

```sql
-- Your query here:
SELECT title, released_year, imdb_rating
FROM movies
WHERE released_year > 2015;
```

Hint: use the WHERE clause to filter your results

A.4 - Find Highly-Rated Movies

Task: The content team wants movies with an IMDB rating of 8.5 or higher. Show `title`, `imdb_rating`, and `no_of_votes`.

```sql
-- Your query here:
SELECT title, imdb_rating, no_of_votes
FROM movies
WHERE imdb_rating >= 8.5;
```

A.5 - Search for a Specific Director

Task: Check if "Christopher Nolan" is in our directors database.

```sql
-- Your query here:
SELECT *
FROM directors
WHERE name = 'Christopher Nolan';
```

Hint: Use `=` for exact matches with text (put text in single quotes)

Section B: Sorting and Limiting Results

Now let's control how our results are displayed.

B.1 - Top Rated Movies

Task: Show the top 10 highest-rated movies. Display `title` and `imdb_rating`, sorted by rating (highest first).

```sql
-- Your query here:
SELECT title, imdb_rating
FROM movies
ORDER BY imdb_rating DESC
LIMIT 10;
```

Hint: DESC = descending (high to low), ASC = ascending (low to high)

B.2 - Most Popular Movies

Task: Find the 5 movies with the most votes. Show `title` and `no_of_votes`, ordered by votes.

```sql
-- Your query here:
SELECT title, no_of_votes
FROM movies
ORDER BY no_of_votes DESC
LIMIT 5;
```

B.3 - Longest Movies

Task: What are the 10 longest movies? Display `title` and `runtime` (in minutes), ordered by `runtime`.

```sql
-- Your query here:
SELECT title, runtime
FROM movies
ORDER BY runtime DESC
LIMIT 10;
```

B.4 - Recent Blockbusters

Task: Find movies from 2010 or later with a rating above 8.0. Show `title`, `released_year`, and `imdb_rating`. Sort by year (newest first).

```sql
-- Your query here:
SELECT title, released_year, imdb_rating
FROM movies
WHERE released_year >= 2010 AND imdb_rating > 8.0
ORDER BY released_year DESC;
```

Hint: Use AND to combine multiple conditions

B.5 - Finding Movies in a Rating Range

Task: Find all movies with ratings between 7.5 and 8.0 (inclusive). Display title and imdb_rating, sorted by rating.

```sql
-- Your query here:
SELECT title, imdb_rating
FROM movies
WHERE imdb_rating >= 7.5 AND imdb_rating <= 8.0
ORDER BY imdb_rating DESC;
```

Alternative: You can use `BETWEEN` instead of `>=` and `<=`

Section C: Using Simple Functions

Let's learn some basic SQL functions to analyze and clean our data.

C.1 - Counting Movies

Task: How many movies are in our database? return the total as `total_movies`

```sql
-- Your query here:
SELECT COUNT(*) AS total_movies
FROM movies;
```

Hint: COUNT(*) counts all rows. AS gives a nickname to the result column.

C.2 - Counting High-Budget Films

Task: How many movies have box office earnings (gross) recorded?

```sql
-- Your query here:
SELECT COUNT(gross) AS movies_with_earnings
FROM movies;
```

Hint: COUNT(column_name) only counts non-NULL values

C.3 - Handling Missing Data

Task: Display movie titles and their meta*score, but show 'No Score' for movies without a meta*score.

```sql
-- Your query here:
SELECT title,
       COALESCE(CAST(meta_score AS TEXT), 'No Score') AS metacritic_score
FROM movies
LIMIT 20;
```

Hint: COALESCE replaces NULL values with something else. check out the sqlite documentation for more info

---

C.4 - Cleaning Up Certificate Data

Task: Show movie titles and certificates. Replace NULL certificates with 'Not Rated'.

```sql
-- Your query here:
SELECT title,
       COALESCE(certificate, 'Not Rated') AS rating_certificate
FROM movies
LIMIT 15;
```

C.5 - Finding Movies with Missing Information

Task: Count how many movies are missing their gross (box office) information.

```sql
-- Your query here:
SELECT COUNT(*) - COUNT(gross) AS movies_without_gross
FROM movies;
```

Hint: COUNT(*) minus COUNT(gross) gives us the count of NULL values

---

Section D: Your First Joins - Connecting Tables

Now for the exciting part! Let's connect tables to answer more complex questions.

D.1 - Movies with Director Names

Task: Show movie titles with their director names (not just director_id). Display the first 10 results.

SQL
```sql
-- Your query here:
SELECT movies.title, directors.name AS director_name
FROM movies
JOIN directors ON movies.director_id = directors.id
LIMIT 10;
```

Hint: JOIN connects two tables (movies and directors) using matching values (director_id = id)

D.2 - Finding a Director's Movies

Task: Find all movies directed by "Christopher Nolan". Show title and released_year.

SQL
```sql
-- Your query here:
SELECT movies.title, movies.released_year
FROM movies
JOIN directors ON movies.director_id = directors.id
WHERE directors.name = 'Christopher Nolan'
ORDER BY movies.released_year;
```

D.3 - Movies and Their Genres

Task: Show movie titles with their genre names. Display the first 20 results.

SQL
```sql
-- Your query here:
SELECT movies.title, genres.name AS genre
FROM movies
JOIN movie_genres ON movies.id = movie_genres.movie_id
JOIN genres ON movie_genres.genre_id = genres.id
LIMIT 20;
```

Hint: You can chain multiple JOINs to connect several tables (movies, movie_genres, genres)

D.4 - Finding Action Movies

Task: Find all movies in the "Action" genre. Display `title` and `imdb_rating`, sorted by rating.

```sql
-- Your query here:
SELECT movies.title, movies.imdb_rating
FROM movies
JOIN movie_genres ON movies.id = movie_genres.movie_id
JOIN genres ON movie_genres.genre_id = genres.id
WHERE genres.name = 'Action'
ORDER BY movies.imdb_rating DESC;
```

D.5 - Movies with Their Lead Actors

Task: Show movies with their lead actors (role_order = 1). Display `movie title` and `actor name`.

```sql
-- Your query here:
SELECT movies.title, actors.name AS lead_actor
FROM movies
JOIN movie_actors ON movies.id = movie_actors.movie_id
JOIN actors ON movie_actors.actor_id = actors.id
WHERE movie_actors.role_order = 1
LIMIT 20;
```

Section E: Combining Everything - Advanced Queries

Let's combine all the concepts you've learned!

E.1 - High-Rated Movies with Directors

Task: Find movies with rating >= 8.5 and show them with director names. Sort by rating.

```sql
-- Your query here:
SELECT movies.title, movies.imdb_rating, directors.name AS director
FROM movies
JOIN directors ON movies.director_id = directors.id
WHERE movies.imdb_rating >= 8.5
ORDER BY movies.imdb_rating DESC;
```

E.2 - Recent Movies in Drama Genre

Task: Find Drama movies from 2010 onwards. Show `title`, `year`, and `rating`.

```sql
-- Your query here:
SELECT movies.title, movies.released_year, movies.imdb_rating
FROM movies
JOIN movie_genres ON movies.id = movie_genres.movie_id
JOIN genres ON movie_genres.genre_id = genres.id
WHERE genres.name = 'Drama'
  AND movies.released_year >= 2010
ORDER BY movies.released_year DESC;
```

E.3 - Director's Best Movies

Task: Find Steven Spielberg's top 5 highest-rated movies. Display `title`, `rating`, and `year`.

```sql
-- Your query here:
SELECT movies.title, movies.imdb_rating, movies.released_year
FROM movies
JOIN directors ON movies.director_id = directors.id
WHERE directors.name = 'Steven Spielberg'
ORDER BY movies.imdb_rating DESC
LIMIT 5;
```

E.4 - Movies with Complete Information

Task: Find movies that have both gross earnings AND meta_score recorded. Show `title` and both values, sorted by gross earnings.

```sql
-- Your query here:
SELECT title, gross, meta_score
FROM movies
WHERE gross IS NOT NULL
  AND meta_score IS NOT NULL
ORDER BY gross DESC
LIMIT 10;
```

Hint: IS NOT NULL checks for non-missing values

E.5 - Star-Studded Movies

Task: Find movies where "Tom Hanks" is the lead actor (role_order = 1). Show `movie title`, `release year`, and `rating`.

```sql
-- Your query here:
SELECT movies.title, movies.released_year, movies.imdb_rating
FROM movies
JOIN movie_actors ON movies.id = movie_actors.movie_id
JOIN actors ON movie_actors.actor_id = actors.id
WHERE actors.name = 'Tom Hanks'
  AND movie_actors.role_order = 1
ORDER BY movies.released_year DESC;
```

Section F: GROUP BY - Aggregating Data

Time to learn one of SQL's most powerful features: GROUP BY. This allows us to create summaries and calculate statistics for groups of data.

Understanding GROUP BY

What does GROUP BY do? GROUP BY groups rows that have the same values in specified columns and allows you to perform calculations on each group.

Think of it like this: - Without GROUP BY: "Count all movies in the database" - With GROUP BY: "Count movies FOR EACH director" or "Count movies FOR EACH genre"

The Rule: When using GROUP BY, you can only SELECT: 1. The columns you're grouping by 2. Aggregate functions (COUNT, AVG, MAX, MIN, SUM) on other columns

Example Structure: `sql SELECT director_id, COUNT(*) as movie_count FROM movies GROUP BY director_id;` This counts how many movies each director has made.

F.1 - Movies Per Director

Task: Count how many movies each director has directed. Show director_id and the count.

```sql
-- Your query here:
SELECT director_id, COUNT(*) AS number_of_movies
FROM movies
GROUP BY director_id
ORDER BY number_of_movies DESC
LIMIT 10;
```

Hint: GROUP BY creates one row per unique director_id

F.2 - Movies Per Year

Task: How many movies were released each year? Show the year and count, sorted by year.

```sql
-- Your query here:
SELECT released_year, COUNT(*) AS movies_count
FROM movies
GROUP BY released_year
ORDER BY released_year DESC;
```

F.3 - Average Rating Per Director

Task: Calculate the average IMDB rating for each director. Show director name and average rating, sorted by average rating (highest first).

```sql
-- Your query here:
SELECT directors.name, AVG(movies.imdb_rating) AS avg_rating
FROM movies
JOIN directors ON movies.director_id = directors.id
GROUP BY directors.id, directors.name
ORDER BY avg_rating DESC
LIMIT 15;
```

Hint: AVG() calculates the average of a numeric column

F.4 - Genre Popularity

Task: Count how many movies belong to each genre. Show genre name and count.

```sql
-- Your query here:
SELECT genres.name AS genre, COUNT(*) AS movie_count
FROM movie_genres
JOIN genres ON movie_genres.genre_id = genres.id
GROUP BY genres.id, genres.name
ORDER BY movie_count DESC;
```

F.5 - Actor Appearances

Task: Count how many movies each actor appears in. Show actor name and count for actors with 5 or more movies.

SQL
```sql
-- Your query here:
SELECT actors.name, COUNT(*) AS movie_appearances
FROM movie_actors
JOIN actors ON movie_actors.actor_id = actors.id
GROUP BY actors.id, actors.name
HAVING COUNT(*) >= 5
ORDER BY movie_appearances DESC;
```

Hint: HAVING filters groups (use it after GROUP BY, while WHERE filters rows before grouping)

F.6 - Directors' Best and Worst

Task: For directors with at least 3 movies, show their name, movie count, highest rating, and lowest rating.

SQL
```sql
-- Your query here:
SELECT directors.name,
       COUNT(*) AS total_movies,
       MAX(movies.imdb_rating) AS best_rating,
       MIN(movies.imdb_rating) AS worst_rating
FROM movies
JOIN directors ON movies.director_id = directors.id
GROUP BY directors.id, directors.name
HAVING COUNT(*) >= 3
ORDER BY best_rating DESC;
```

Hint: You can use multiple aggregate functions in one query

F.7 - Box Office by Genre

Task: Calculate the total and average box office (gross) for each genre. Only include genres with at least 10 movies.

SQL
```sql
-- Your query here:
SELECT genres.name AS genre,
       COUNT(*) AS movie_count,
       SUM(movies.gross) AS total_gross,
       AVG(movies.gross) AS avg_gross
FROM movies
JOIN movie_genres ON movies.id = movie_genres.movie_id
JOIN genres ON movie_genres.genre_id = genres.id
WHERE movies.gross IS NOT NULL
GROUP BY genres.id, genres.name
HAVING COUNT(*) >= 10
ORDER BY total_gross DESC;
```

Hint: SUM() adds up all values in a group

🎉 Congratulations!

You've completed your first SQL worksheet! You've learned how to: - Write basic SELECT queries - Filter results with WHERE - Sort with ORDER BY and limit results - Use functions like COUNT and COALESCE - Join tables to answer complex questions - Use GROUP BY to aggregate and summarize data - Filter groups with HAVING

Final Challenge: Write a query to find which year had the highest average IMDB rating for movies (considering only years with at least 5 movies).

Quick Reference

- **SELECT**: Choose columns to display
- **FROM**: Specify the table
- **WHERE**: Filter rows (before grouping)
- **JOIN...ON**: Connect tables
- **GROUP BY**: Group rows for aggregation
- **HAVING**: Filter groups (after grouping)
- **ORDER BY**: Sort results
- **LIMIT**: Restrict number of rows

Aggregate Functions: - **COUNT()**: Count rows - **AVG()**: Calculate average - **SUM()**: Add up values - **MAX()**: Find maximum value - **MIN()**: Find minimum value - **COALESCE()**: Replace NULL values - **AND/OR**: Combine conditions - **IS NULL / IS NOT NULL**: Check for missing values