IMDB Mini-Database Project Report

Course: Data Management

Project: Relational Database Design and Analysis

Names:

Hawa Shams

Bahar Shafieian

Mubina Beknazarova

Date: December 2025

Introduction

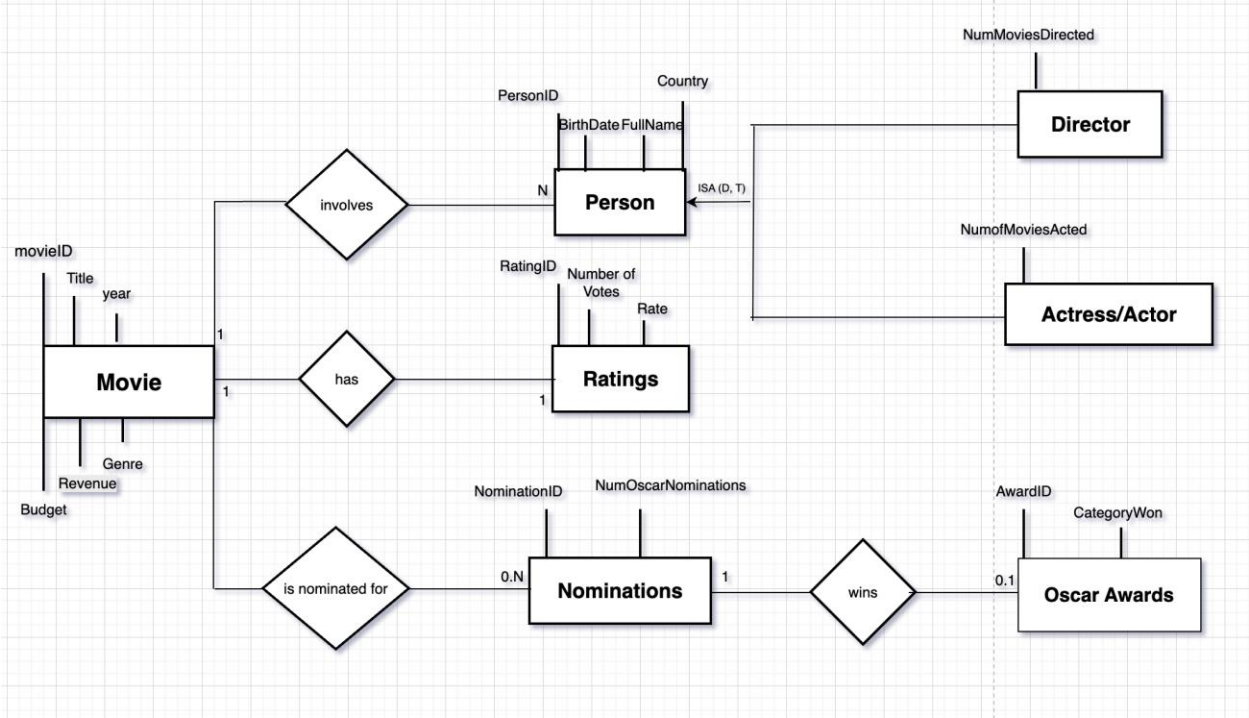
This project focuses on the design, implementation, and analysis of a simplified IMDb-style relational database system using MySQL.
The goal was to model essential movie-related information—Titles, directors, actors, ratings, nominations, and Oscar awards—and perform analytical queries to extract insights.
Python was also used to generate visualizations that complement the SQL analysis.

Conceptual & Logical Design

A structured ER model was developed containing the following entities:

Entities

- **Movie** (movieID, Title, Year, Genre, Country, Budget, Revenue)
- Person (PersonID, BirthDate, FullName, Country)
- **Director** (NumMoviesDirected)
- **Actor** (NumMoviesActed)
- **Ratings** (RatingID, Number of Votes, Rate)
- **Nominations** (NominationID, NumberOscarNominations)
- **Oscar Awards** (AwardID, CategoryWon)



Relationships

Although a real IMDb-scale database would typically require **many-to-many relationships** (e.g., many actors per movie, many directors per movie), our project intentionally uses **simplified cardinalities** for two reasons:

1. **Project scope and data availability**
2. **Requirement to keep the dataset small and controlled (≈10 movies)** In our conceptual model, the **Movie** entity is central and connects to several other entities through well-defined relationships.

Movie → Person (1:N)

A movie typically involves multiple people (such as directors and actors). This is represented as a **1-to-N** relationship:

- **One movie** can be associated with **many persons**.
- In our dataset, the *Person* Super Entity is later specialized into Director and **Actor/Actress**, allowing us to distinguish roles while maintaining a unified representation at the conceptual level.

ISA = (D, T)

In our dataset specifically:

- Disjoint (D) → A person cannot be both Actor and Director in our dataset
- Total (T) → Every Person must be either Actor or Director, since we are not including writers, producers, etc.

Movie → Ratings (1:1)

For this project, we considered only **one type of rating**, namely the IMDb score.
Therefore, each movie is associated with **exactly one rating entry**, resulting in a **1-to-1** relationship.
This simplification keeps the design aligned with the dataset, which provides a single aggregated rating per movie.

Movie → Nominations (1.0.N

A movie may receive **multiple nominations** or **none**.
This is expressed through a **0.N cardinality** on the Nominations side:

- Some movies are never nominated (0).
- Others may receive several nominations (N).
  This flexibility accurately reflects the award process.

Nominations → Oscar_Awards (1.0.1)

A nomination may result in **zero or one** Oscar award:

- Zero, if the movie is nominated but does not win.
- One, if the nomination leads to an Oscar win.
  This is captured by 0.1 participation on the Oscar_Awards side.
  Each Oscar award corresponds to a specific nomination event, ensuring correctness in how wins are recorded.

Specialization in Our Design

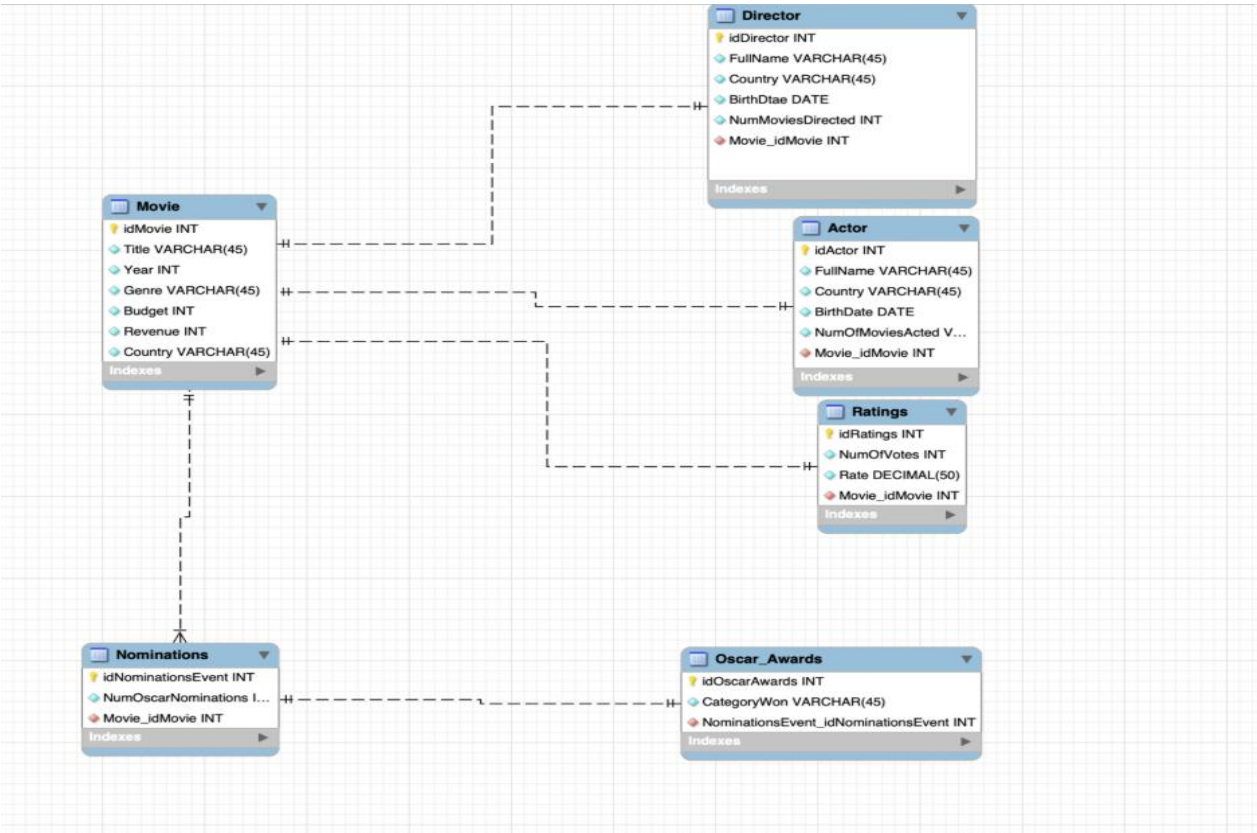
In the ER model, **Person** is a super-entity that specializes in **Actor** and **Director**. Each subtype has the shared person's attributes (FullName, Country, BirthDate) plus its own

specific attributes. Although the relational schema does not include a separate Person table.

Rationale Behind Entity and Attribute Selection:

We selected entities according to whether a concept represents a “thing” in the application domain (with its own identity and facts) or just a descriptive label.

- Movie, Actor, Director, Ratings, Nominations, and Oscar_Awards were modelled as entities because they correspond to concrete objects or events we want to count, list, and describe independently (e.g. an actor, a nomination event, a rating).
- On the other hand, Genre and the Oscar CategoryWon were kept as simple attributes. In the scope of this project, they are only labels that classify a movie or an award; we do not store additional information about genres or categories, nor do we have separate relationships involving them. Promoting them to entities would add tables without adding modelling power for the analyses we perform.



In the logical schema we decided not to materialize *Person* as a separate table. Instead, we implemented the ISA hierarchy by creating two independent tables, Actor and Director, each containing the “Person” attributes plus their role-specific fields.

This choice simplifies the implementation and the queries of interest for the project, since we mainly access actors and directors separately, and we do not need to frequently query "all persons regardless of their role". When necessary, the conceptual set of persons can still be reconstructed as the union of the two tables (using JOIN).

Each specialized table inherits the common attributes of a person — such as **FullName**, **Country**, and **BirthDate** — while also including role-specific attributes (e.g., number of movies acted or directed).
This approach simplifies queries and maintains data clarity while respecting the structure of the original ER model.

Tables and Their Attributes

Movie

In our conceptual schema, **Movie** represents the central entity, since the primary information managed by the database concerns films. The other entities are defined mainly to describe different aspects and events related to movies. For this reason, Movie can be considered the core table of the design, even though, from a relational point of view, all tables are on the same level.

The **Movie** table stores core descriptive and financial information about each film.

- **IdMovie** (INT, PK)
- **Title** (VARCHAR)
- **Year** (INT)
- **Genre** (VARCHAR)
- **Budget** (INT)
- **Revenue** (INT)
- **Country** (VARCHAR)

Each movie connects to ratings, nominations, a primary actor, and a primary director.

Data Preparation and Construction of the Final Dataset

To get the final dataset we did the following steps:

- ***Data integration.***

  We started from two separate IMDB CSV files (tmdb_5000_movies.csv and tmdb_5000_credits.csv) and loaded them in Python. Using the common movie identifier (id in *movies*, movie_id in *credits*), we renamed movie_id to id and merged the two datasets into a single DataFrame containing both movie metadata and cast/crew information. Afterwards, we integrated external information collected manually (date of birth and country for the main actor and director, number of Oscar nominations and Oscar wins for each film) by adding these fields as new columns to the same dataset.

- ***Data cleaning***

  On the merged data, we cleaned and filtered the information. We converted release_date from string to a proper datetime type, created a year column, and kept only movies released between 1990 and 2000. We then selected only the columns needed for the project (title, year, rating, number of votes, genres, actor, director, budget, revenue, etc.) and safely parsed the JSON-like text columns (genres, actor, director) so that invalid rows would not break the code. Finally, we filtered the dataset to movies with vote_average > 7.5 and, among these, kept only the 10 most popular titles by sorting on vote_count and taking the top 10.

5 rows × 23 columns

```
## 3. Convert Release Date and Extract the Release Year
```

```
df["release_date"] = pd.to_datetime(df["release_date"], errors="coerce")
df["year"] = df["release_date"].dt.year
```

```
## 4. Filter Movies Released Between 1990 and 2000
```

```
df_filtered = df[(df["year"] >= 1990) & (df["year"] <= 2000)]
df_filtered[["title", "year"]].head()
```

	title	year
25	Titanic	1997.0
70	Wild Wild West	1999.0
75	Waterworld	1995.0
149	Armageddon	1998.0
164	Lethal Weapon 4	1998.0

```
## 5. Select Columns Needed for Cleaning
```

```
df_clean = df_filtered[[
    "title", "year", "vote_average", "vote_count",
    "genres", "cast", "crew", "budget", "revenue"
]].copy()
df_clean.head()
```

	title	year	vote_average	vote_count	genres	cast	crew	budget	revenue
25	Titanic	1997.0	7.5	7562	[{"id": 18, "name": "Drama"}, {"id": 10749, "n...	[{"cast_id": 20, "character": "Rose DeWitt Buk...	[{"credit_id": "52fe425ac3a36847f8017985", "de...	200000000	1845034188

- ***Data transformation***

  We transformed several columns to make them compatible with our relational model. From the crew JSON we extracted, for each movie, the director; from the

cast JSON we extracted only the main character (first actor) to avoid having lists of actors in a single cell, and we applied the same idea to genres by keeping only the main genre. integer IDs for movies, actors and directors, and counts of how many movies each actor/director appears in within our sample. At the end, we reordered the columns into a logical structure and exported the final 10×21 table to CSV.

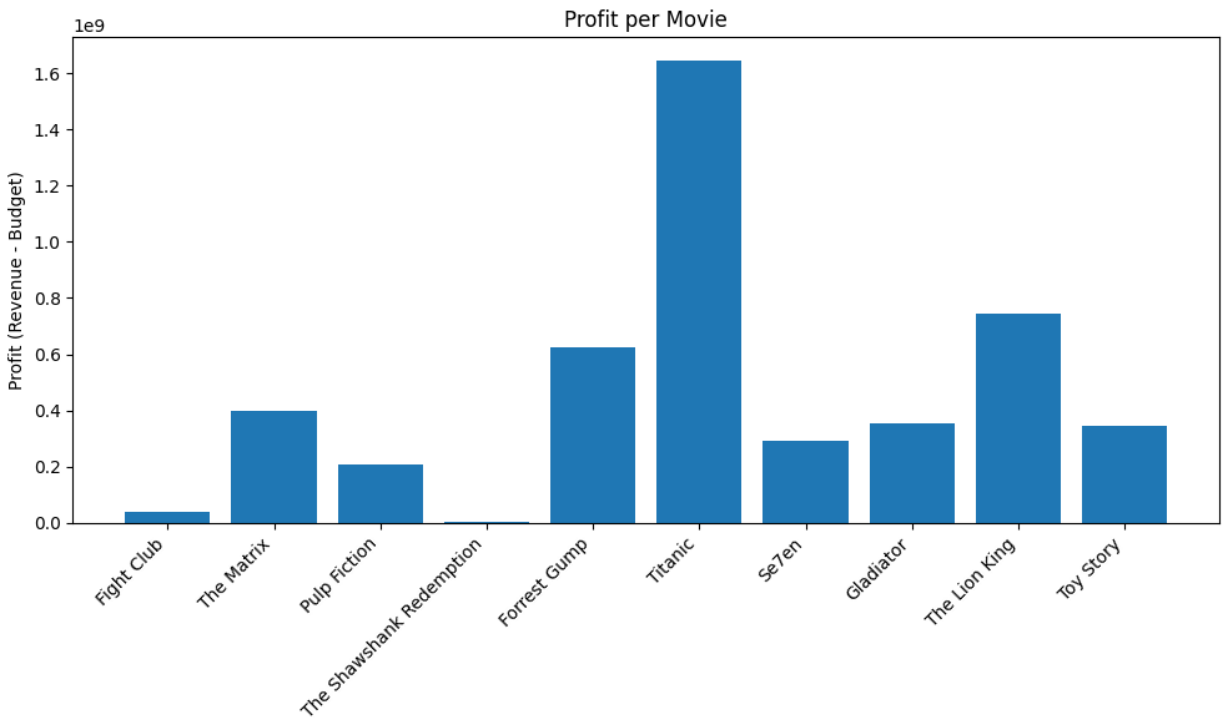
- *Data analysis.*
  During these steps we also carried out simple exploratory analysis to guide our choices. We used the rating (vote_average) and popularity (vote_count) to identify the "best 10 movies of the decade", and the counts movies_acted and movies_directed to summarise how active each main actor and director is in our mini-IMDB. These analytical checks helped us justify the final selection of movies and variables that we then used to build and populate the SQL database.

- ***Data Visualization***

For the visual analysis of financial performance and award outcomes, we used Python to extract data directly from the MySQL database and generate two plots. The visualization code relies on three main libraries:

- **pandas**: for loading SQL tables as DataFrames and performing data manipulation (profit calculation, grouping wins, merging tables).
- **SQLAlchemy** (with the **pymysql** driver): for establishing a connection to the MySQL database and executing SQL queries.
- **matplotlib.pyplot**: for constructing bar charts and scatter plots to visualize profit, nominations, and wins.

Using these libraries, we connected to the IMDB database, retrieved the Movie, Ratings, Nominations, and Oscar_Awards tables, computed new analytical variables (profit per movie and wins per movie), and produced two plots: a bar chart of movie profits and a scatter plot showing the relationship between Oscar nominations and wins.



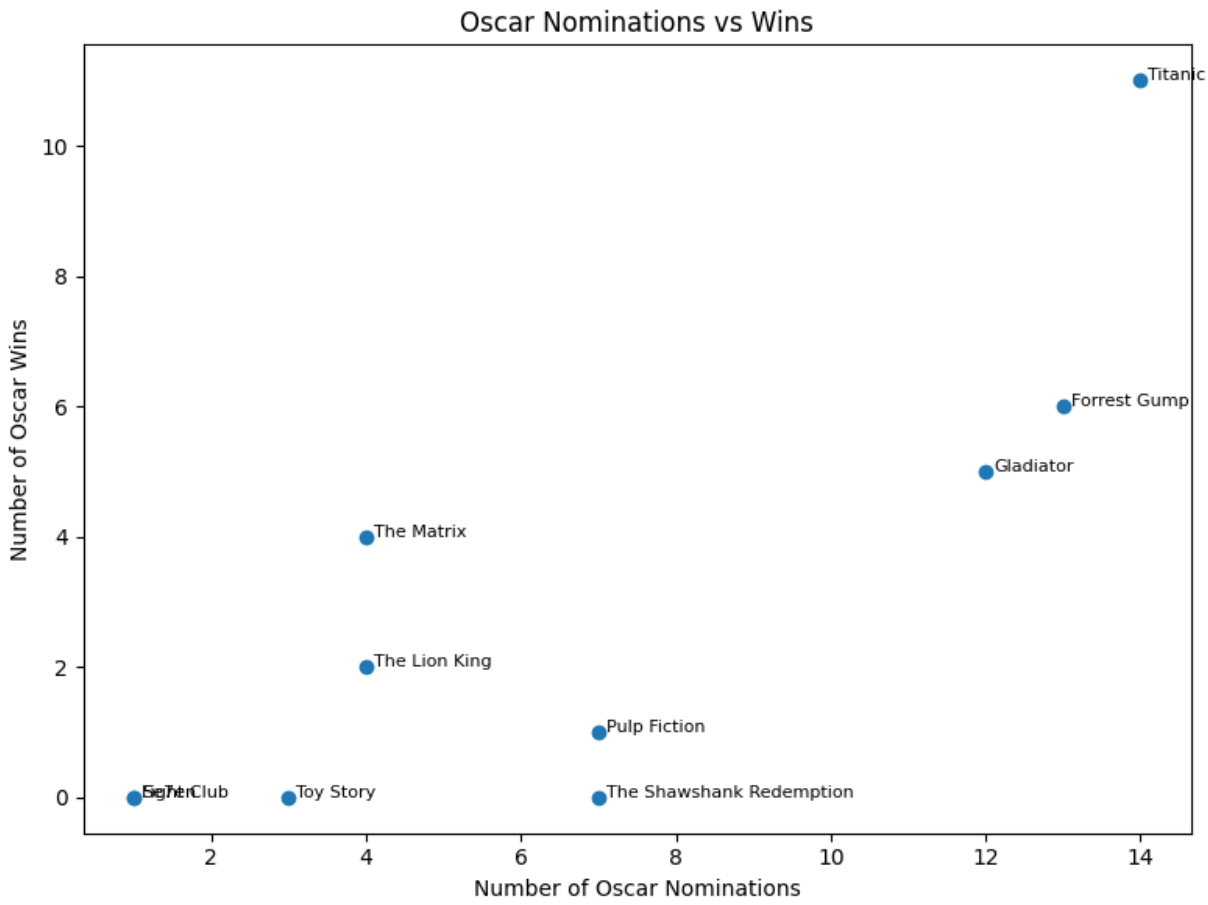
1. Profit per Movie

This bar chart compares the financial performance of the ten selected movies using the metric:

Profit = Revenue – Budget

Titanic is the most profitable movie by a large margin, exceeding 1.6 billion USD in profit. Forrest Gump, The Lion King, Gladiator, and The Matrix also show very high profitability. Movies like Fight Club and The Shawshank Redemption show much smaller profit margins, indicating that commercial success did not always match critical success.

This visual supports our Data Analysis step by showing financial performance differences across the movies selected for SQL database and demonstrates why Budget and Revenue were included as a derived attribute in some views.



2. Oscar Nominations vs Wins

This scatter plot shows the relationship between:

- **Number of Oscar nominations** (x-axis)
- **Number of Oscar wins** (y-axis)

Titanic clearly dominates with 14 nominations and 11 wins. Forrest Gump and Gladiator also display strong award performance. Some films such as The Shawshank Redemption received many nominations but won none, while others like The Matrix achieved a high win-to-nomination ratio.

We Used throughout all steps for data loading, cleaning, transformation, Visulaization and export:

pandas

We Used throughout all steps for data loading, cleaning, transformation, and export.
Key functions:

- pd.read_csv() – load IMDB datasets
- df.merge() – merge movie and credits tables
- df.rename() – rename columns **data transforamtion**
- df.drop() and column selection – keep only relevant fields
- pd.to_datetime() – convert release dates **data cleaning**
- df.sort_values() – select top-10 movies by vote_count
- df.groupby().transform("count") – count movies acted/directed
- df.apply() – extract director and main actor
- df.to_csv() – export final dataset

numpy

- Used for simple numerical operations when needed.
  Key function:
- np types and basic math operations.

ast (Abstract Syntax Trees)

- Used to safely parse JSON-like strings from the IMDB dataset.
  Key function:
- ast.literal_eval() – convert text into Python lists/dictionaries for genres, cast, crew.

SQLAlchemy + PyMySQL

- Used only in the visualization stage to query the MySQL database.
  Key functions:
- create_engine() – create database connection
- pd.read_sql() – load MySQL tables into pandas

matplotlib.pyplot

- Used to generate the two visualizations (profit bar chart, nominations vs wins scatter plot).
  Key functions:
- plt.bar() – profit per movie
- plt.scatter() – nominations vs wins
- plt.text() – annotate points
- plt.tight_layout() – clean formatting
- plt.show() – display plots

Database Population (SQL)

The database was populated through a dedicated SQL population script built to fully respect the logical schema and all relational integrity constraints. At the beginning of execution, the script selects the correct schema using USE IMDB; and resets all tables by temporarily disabling referential integrity with SET FOREIGN_KEY_CHECKS = 0;. This allows the script to run DELETE statements in reverse foreign-key order (Oscar Awards → Nominations → Ratings → Actor → Director → Movie) without violating dependencies. After the cleanup, SET FOREIGN_KEY_CHECKS = 1; re-enables constraint enforcement, ensuring that all subsequent inserts must satisfy the declared FK relationships.

Population follows the correct **foreign key hierarchy**:

- INSERT INTO Movie populates all parent rows first.
- INSERT INTO Director and INSERT INTO Actor load tables that reference Movie via FK fields.
- ALTER TABLE Ratings MODIFY … is used to fix the DECIMAL type before inserting rating values……

All inserted values were selected to be realistic with respect to data types and domain constraints (valid dates, revenues > 0, ratings in the 1–10 range, consistent foreign keys). The dataset is intentionally heterogeneous, containing movies with different genres, years, revenues, and award histories so that analytical queries (joins, aggregations, filters, nested queries) produce meaningful patterns.

The script is modular, structured into labeled sections, uses explicit INSERT INTO … VALUES … blocks, and is fully re-runnable from scratch because the cleanup stage ensures a deterministic starting state. This makes the population process transparent, reproducible, and maintainable.

Queries

```
50      -- ============================================
51      -- 4. Average Rating per Genre
52      -- ============================================
53  •  SELECT M.Genre, AVG(R.Rate) AS AvgRating
54      FROM Movie M
55      JOIN Ratings R ON M.idMovie = R.Movie_idMovie
56      GROUP BY M.Genre
57      ORDER BY AvgRating DESC;
58
```

13:54

esult Grid | Filter Rows: Search | Export:

Genre	AvgRating
Thriller	8.30000
Comedy	8.20000
Drama	8.10000
Crime	8.10000
Family	8.00000
Action	7.90000
Animation	7.70000

This query computes the **average IMDb rating** for each movie genre in the database. The query performs the following steps:

1. **Join Movie and Ratings** through the foreign key Movie_idMovie, ensuring that each rating is matched with the correct movie.
2. **Group the results by Genre**, so each genre appears once in the output.
3. **Apply the aggregate function AVG(R.Rate)** to compute the mean rating per genre.
4. **Order by AvgRating DESC** to highlight the highest-rated genres first.

```sql
-- ========================================
-- 2. Oscar Categories Per Movie (Grouped)
-- ========================================
SELECT
    M.Title,
    GROUP_CONCAT(
        O.CategoryWon
        ORDER BY O.CategoryWon
        SEPARATOR ', '
    ) AS CategoriesWon
FROM Movie AS M
JOIN Nominations AS N
  ON M.idMovie = N.Movie_idMovie
JOIN Oscar_Awards AS O
  ON O.NominationsEvent_idNominationsEvent = N.idNominationsEvent
GROUP BY M.idMovie, M.Title
ORDER BY M.Title;
```

00% 12:27

Result Grid | Filter Rows: Search | Export:

Title	CategoriesWon
Forrest Gump	Best Actor, Best Director, Best Editing, Best Effects, Best Picture, Best Score
Gladiator	Best Actor, Best Costume, Best Effects, Best Picture, Best Sound
Pulp Fiction	Screenplay
The Lion King	Best Score, Best Song
The Matrix	Technical Award, Technical Award, Technical Award, Technical Award
Titanic	Best Art Direction, Best Cinematography, Best Costume, Best Director, Best Editing, Best Effects, Best Makeup, Best Picture, Best Score, Best Song, Best Sound

This query produces a consolidated list of all Oscar categories won by each movie. It integrates information from **three related tables**—Movie, Nominations, and Oscar_Awards—using foreign-key joins to reconstruct the full award history of each film.

Group By: groups rows that share the same value(s) so you can apply aggregate functions (like COUNT, SUM, AVG, MIN, MAX) per group, instead of over the whole table.

1. **Join Movie → Nominations**
   Matches every movie with its corresponding Oscar nomination record.
2. **Join Nominations → Oscar_Awards**
   Retrieves every award category actually won by that nomination event.
3. **GROUP_CONCAT(CategoryWon)**
   Aggregates multiple award rows into a **single comma-separated string**, making the result easy to read—for example:
   "Best Picture, Best Director, Best Editing".
4. **GROUP BY movie**
   Ensures that each movie appears once, even if it has many Oscar wins.
5. **ORDER BY Title**
   Produces alphabetically ordered output for readability.

In this query, GROUP BY is used to aggregate the results per movie. Because each movie can have multiple Oscar_Awards rows (one for each category won), the joins would normally produce one row per category. By grouping on M.idMovie, M.Title, we tell SQL to produce one output row for each movie, and then use GROUP_CONCAT to merge all the categories won into a single comma-separated string (sorted alphabetically). So the final output is: one row per movie + a list of all categories it won.

Conclusion

This project demonstrates the complete lifecycle of designing and implementing a relational movie database, from conceptual modeling to data preparation, SQL population, analytical querying, and visualization. By integrating information from the IMDB datasets and enriching it with manually collected metadata, we created a coherent 10-movie mini-IMDB that highlights the interaction between movies, people, ratings, nominations, and awards. The logical schema, combined with a controlled and reproducible population script, ensured data consistency and meaningful analytical output. The SQL queries and Python visualizations provided insights into financial performance, and award outcomes, validating the design choices made throughout the project. Overall, the project shows how relational modeling, data transformation, and analytical querying can work together to support a compact but expressive movie information system.