Movie Data Analysis

— Project Report —

Bilal Tanvir Bhatti

June 30, 2024

Contents

1 Introduction

This report presents a detailed analysis of a movie dataset as part of a coding challenge for the role of Product Analyst. The primary objective is demonstrating the ability to connect to a database, import and manipulate data using SQL, and generate insightful visualizations. The dataset consists of three main tables—titles, ratings, and credits—which provide comprehensive information about movies, including release years, IMDb ratings, and the actors involved.

In this project, we completed multiple analytical tasks including removing duplicate records, studying trends in movie runtimes, and investigating the distribution of IMDb scores within different voting ranges. Furthermore, the report examines movie genres and actor involvement over time. Tasks are supported by SQL code, explanations, and visual representations to demonstrate the results. This project highlights using SQL and data visualization abilities to derive valuable insights from a well-organized dataset.

In Section 2, insights about database tables and data import are given. Section 3, explains the tasks and the analysis. Lastly, the complete project is summarised in 4 section.

2 Methodology

2.1 Dataset Description

The source data for this analysis can be found in the `movies` folder. It consists of three tables: `titles`, `ratings`, and `credits`. Each table provides different aspects of information about movies and TV shows. Below is a detailed description of each table and the variables they contain.

2.1.1 Titles

The `titles` table includes a comprehensive list of movies and TV shows along with their respective details. The primary key of this table is the `id` column. This table contains the following variables:

- **id**: Unique identifier for each movie or TV show.
- **title**: The title of the movie or TV show.
- **type**: Specifies whether the entry is a movie or a TV show.
- **description**: A brief description of the movie or TV show.
- **release_year**: The year the movie or TV show was released.

- **age_certification**: The age certification rating of the movie or TV show.
- **runtime**: The duration of the movie or TV show in minutes.
- **genres**: The genres associated with the movie or TV show.
- **production_countries**: The countries where the movie or TV show was produced.
- **seasons**: The number of seasons, applicable if the entry is a TV show.
- **imdb_id**: The IMDb identifier for the movie or TV show.
- **imdb_score**: The IMDb rating score for the movie or TV show.

2.1.2 Ratings

The `ratings` table contains IMDb ratings for movies and TV shows, partially overlapping with the `titles` table. The primary key for this table is the `id` column. This table includes the following variables:

- **index**: The index of the entry in the dataset.
- **id**: Unique identifier that links to the `id` column in the `titles` table.
- **title**: The title of the movie or TV show.
- **type**: Specifies whether the entry is a movie or a TV show.
- **description**: A brief description of the movie or TV show.
- **release_year**: The year the movie or TV show was released.
- **age_certification**: The age certification rating of the movie or TV show.
- **runtime**: The duration of the movie or TV show in minutes.
- **imdb_id**: The IMDb identifier for the movie or TV show.
- **imdb_score**: The IMDb rating score for the movie or TV show.
- **imdb_votes**: The number of votes the movie or TV show has received on IMDb.

2.1.3 Credits

The `credits` table lists actors and directors associated with the movies and TV shows from the `ratings` and `titles` tables. The `id` column in this table links each person to the respective movie or TV show. This table contains the following variables:

- **person_id**: Unique identifier for each person (actor or director).
- **id**: Unique identifier that links to the `id` column in the `titles` and `ratings` tables.
- **name**: The name of the actor or director.
- **character**: The character played by the actor in the movie or TV show.
- **role**: Specifies the role of the person, either actor or director.

These tables collectively provide a rich dataset for analyzing various aspects of movies and TV shows, including their details, ratings, and the people involved in their production.

2.2 Data Import and Setup

The process of setting up the database and importing data involves several key steps. This section describes the configuration, database setup, database loading, and the scripts used to manage these operations.

2.2.1 Configuration

The configuration settings are stored in the `config.py` file, which defines the database name and the path to the SQL initialization script. The contents of the `config.py` file are as follows:

```
db_name = './database/movie-analysis.db'
sql_file_path = './src/db/init.sql'
```

2.2.2 Database Setup

The main database setup and data import operations are handled in the `main.py` file. The `db_setup` function orchestrates the entire process:

```
def db_setup():
    # Connecting database
    conn = create_and_connect_database(db_name)

    # Emptying the database
```

```
    drop_all_tables(conn)

    # Inserting tables in db
    sql_script = read_sql_file(sql_file_path)
    execute_sql_script(conn, sql_script)

    # Loading data into the tables
    load_csv_data(conn, "./movies/credits.csv", "credits")
    load_csv_data(conn, "./movies/titles.csv", "titles")
    load_csv_data(conn, "./movies/ratings.csv", "ratings")

    # DB connection closed
    conn.close()
    print("Database␣connection␣closed.")
```

2.2.3 Database Functions

The `db.py` file contains all the functions required for creating the database, reading SQL scripts, executing SQL commands, loading CSV data into the database, and dropping existing tables. These functions are as follows:

```
import sqlite3
import csv

def create_and_connect_database(db_name):
    try:
        conn = sqlite3.connect(db_name)
        print(f"Database␣'{db_name}'␣created␣successfully.")
        return conn
    except Exception as e:
        print(f"DB␣creation␣error:␣{e}")

def read_sql_file(file_path):
    try:
        with open(file_path, 'r') as file:
            sql_script = file.read()
            print(f"SQL␣file␣'{file_path}'␣read␣successfully.")
            return sql_script
    except Exception as e:
        print(f"SQL␣file␣reading␣error:␣{e}")

def execute_sql_script(conn, sql_script):
    try:
        with conn:
            conn.executescript(sql_script)
        print("SQL␣script␣executed␣successfully.")
    except sqlite3.Error as e:
        print(f"An␣error␣occurred:␣{e}")

def load_csv_data(conn, csv_file, table_name):
    try:
```

```
        cursor = conn.cursor()
        with open(csv_file, 'r', encoding='utf-8') as file:
            reader = csv.DictReader(file)

            columns = ', '.join(reader.fieldnames)
            placeholders = ', '.join('?' * len(reader.fieldnames))
            sql = f'INSERT INTO {table_name} ({columns}) VALUES ({
                placeholders})'

            for row in reader:
                cursor.execute(sql, list(row.values()))

        conn.commit()
    except Exception as e:
        print(f"CSV loading error: {e}")

def drop_all_tables(conn):
    try:
        cursor = conn.cursor()

        # Get list of tables in the database
        cursor.execute("SELECT name FROM sqlite_master WHERE type='
            table';")
        tables = cursor.fetchall()

        # Drop each table
        for table in tables:
            table_name = table[0]
            cursor.execute(f"DROP TABLE IF EXISTS {table_name};")
            print(f"Table '{table_name}' dropped successfully.")

        conn.commit()
    except Exception as e:
        print(f"An error occurred: {e}")
```

2.2.4 Process Overview

1. **Connecting to the Database**: The `create_and_connect_database` function establishes a connection to the SQLite database specified in the configuration file. If the database does not exist, it is created.

2. **Emptying the Database**: The `drop_all_tables` function removes all existing tables from the database to ensure a clean slate for data import.

3. **Executing SQL Script**: The `read_sql_file` function reads the SQL initialization script from the specified file path. The `execute_sql_script` function then executes this script to create the necessary tables in the database.

4. **Loading CSV Data**: The `load_csv_data` function imports data from CSV files into the corresponding tables in the database. This function reads the CSV files for the

`credits`, `titles`, and `ratings` tables and inserts the data into the database.

5. **Closing the Connection**: Finally, the database connection is closed, ensuring all operations are properly completed and resources are released.

By following this process, the database is set up and populated with the necessary data, ready for further analysis and querying.

3 Tasks and Analysis

3.1 Task 1: Deduplicated List of Movies and Shows

3.1.1 Objective

The objective of this task is to create a deduplicated list of movies and shows by combining information from the `titles` and `ratings` tables. The resulting table should include all fields that both tables have in common, ensuring that each movie is listed only once.

3.1.2 SQL Code

```
CREATE TABLE deduplicated_list AS
SELECT
    t.id AS id,
    t.title AS title,
    t.type AS type,
    t.description AS description,
    t.release_year AS release_year,
    t.age_certification AS age_certification,
    t.runtime AS runtime,
    r.imdb_id AS imdb_id,
    r.imdb_score AS imdb_score,
    r.imdb_votes AS imdb_votes
FROM
    titles t
JOIN
    ratings r ON t.imdb_id = r.imdb_id
GROUP BY
    t.id;
```

3.1.3 Explanation

The SQL query performs the following steps to achieve the objective:

- **CREATE TABLE deduplicated_list AS**: This statement creates a new table named `deduplicated_list` to store the results of the query.

- **SELECT**: The SELECT clause specifies the columns to be included in the resulting table. The columns are selected from both the `titles` (aliased as `t`) and `ratings` (aliased as `r`) tables.
- **t.id, t.title, t.type, t.description, t.release_year, t.age_certification, t.runtime**: These columns are selected from the `titles` table, providing information about the movie or show.
- **r.imdb_id, r.imdb_score, r.imdb_votes**: These columns are selected from the `ratings` table, adding IMDb-related information to the result.
- **FROM titles t JOIN ratings r ON t.imdb_id = r.imdb_id**: This JOIN clause combines rows from the `titles` and `ratings` tables where the `imdb_id` values match, ensuring that only movies and shows with corresponding ratings are included.
- **GROUP BY t.id**: The GROUP BY clause groups the results by the movie or show ID, effectively deduplicating the entries.

3.1.4 Results

The resulting table, `deduplicated_list`, contains a unique list of movies and shows with comprehensive information from both the `titles` and `ratings` tables. This table forms the basis for further analysis in subsequent tasks.

3.2 Task 2: Longest and Shortest Runtimes Among Movies

3.2.1 Objective

The objective of this task is to find the longest and shortest runtimes among the movies in the deduplicated list created in Task 1.

3.2.2 SQL Queries

Longest Runtimes Query:

```
SELECT MAX(runtime) AS
   Longest_runtime
FROM deduplicated_list
WHERE type = 'MOVIE';
```

Shortest Runtimes Query:

```
SELECT MIN(runtime) AS
   Shortest_runtime
FROM deduplicated_list
WHERE type = 'MOVIE';
```

3.2.3 Explanation

The SQL queries perform the following steps to achieve the objective:

- **Longest Runtimes Query**: Finds the maximum runtime (`MAX(runtime)`) among movies by filtering rows where the `type` is 'MOVIE'.
- **Shortest Runtimes Query**: Finds the minimum runtime (`MIN(runtime)`) among movies by filtering rows where the `type` is 'MOVIE'.

3.2.4 Results

After executing the queries:

- The longest runtime among movies is 225 minutes.
- The shortest runtime among movies is 2 minutes.

These results provide insights into the range of runtimes for movies in the dataset.

3.3 Task 3: Average Runtime per Release Year Analysis

3.3.1 Objective

The objective of this task is to analyze whether there is an increase or decrease in the average runtime per release year for movies and shows separately.

3.3.2 SQL Query

```
SELECT release_year, runtime, "type"
FROM    deduplicated_list
WHERE release_year IS NOT NULL AND runtime IS NOT NULL
```

3.3.3 Python Processing

The following Python code processes the data to calculate average runtimes for movies and shows per release year:

```
# Filtering the data w.r.t MOVIE and SHOW
movies_df = df[df['type'] == 'MOVIE']
shows_df = df[df['type'] == 'SHOW']


movies_avg_runtime = movies_df.groupby('release_year')['runtime'].mean
    ().reset_index()
shows_avg_runtime = shows_df.groupby('release_year')['runtime'].mean().
    reset_index()
```

The last two lines of Python code process the filtered dataframes, ‘movies_df‘ and ‘shows_df‘, to calculate the average runtime per release year for movies and shows respectively.

The line ‘movies_avg_runtime = movies_df.groupby(’release_year’)[’runtime’].mean()‘ groups the ‘movies_df‘ dataframe by the ‘release_year‘ column. For each unique release year, it selects the ‘runtime‘ column values, calculates their mean (average), and returns a series object where each release year is paired with its corresponding average runtime.

Similarly, ‘shows_avg_runtime = shows_df.groupby(’release_year’)[’runtime’].mean()‘ performs the same operation on the ‘shows_df‘ dataframe, computing the average runtime per release year for TV shows.

To convert these series objects back into proper dataframes, ‘reset_index()‘ is appended to each line. This method resets the index of the series, turning the release years and their respective average runtimes into columns within a new dataframe. Thus, ‘movies_avg_runtime‘ and ‘shows_avg_runtime‘ become dataframes suitable for further analysis and visualization in Python or for exporting into other formats.

3.3.4 Results

The results are visualized in the Figure 1:

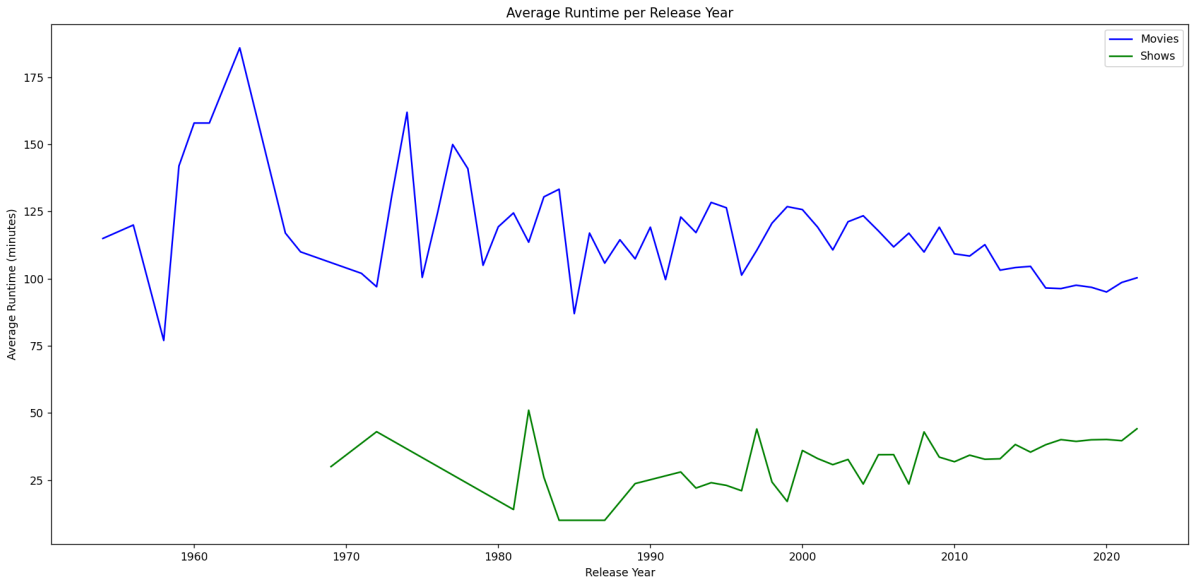

Figure 1: Average Runtime per Release Year

3.3.5 Interpretation

The Figure 1 titled “Average Runtime per Release Year” compares the average runtime trends of movies and shows from approximately 1960 to 2020:

- **For Movies (Blue Line)**: - The average runtime starts around 125 minutes in 1960. - There are significant fluctuations over the years, with peaks reaching nearly 175 minutes and troughs dropping below 100 minutes. - Overall, there's a gradual decline in average runtime, ending just above 100 minutes by 2020.
- **For Shows (Green Line)**: - The average runtime starts just above zero in 1960. - The runtime steadily increases over the decades, with minor variations. - By 2020, the average runtime for shows reaches around 50 minutes.

Movies generally exhibit higher runtimes compared to shows. Shows have seen a steady increase in runtime, whereas movies have shown more variability and a slight overall decrease.

3.4 Task 4: Analysis of Title Lengths

3.4.1 Objective

The objective of this task is to identify the 10 longest and 10 shortest titles from the `deduplicated_list` table and display their lengths.

3.4.2 SQL Queries

Longest Titles

```
SELECT title, LENGTH(title) AS
   title_length
FROM deduplicated_list
ORDER BY title_length DESC
LIMIT 10;
```

Shortest Titles

```
SELECT title, LENGTH(title) AS
   title_length
FROM deduplicated_list
ORDER BY title_length ASC
LIMIT 10;
```

3.4.3 Results

The Table 1 and Table2 show the Longest and shortest titles.

Table 1: Longest Titles

Title	Length
Jim and Andy: The Great Beyond - Featuring a Very Special, Contractually..	104
One Piece: Episode of Chopper Plus: Bloom in Winter, Miracle Sakura	79
Road to El Dorado: America's Secret Space Escapes	75
The Judah Friedlander: America is the Greatest Country in the United States	71
LEGO Marvel Super Heroes: Guardians of the Galaxy - The Thanos Threat	69
One Piece: Desert Princess and the Pirates Adventures in Alabasta	68
The Woman in the House Across the Street from the Girl in the Window	68
Here Comes Peter Cottontail: The Movie	64
The 108-year-old Man Who Skipped Out on the Bill and Disappeared	64
The Crystal Calls - Making The Dark Crystal: A...	63

Table 2: Shortest Titles

Title	Length
H	1
PK	2
42	2
IO	2
21	2
83	2
%2	2
One	3
UFO	3
11M	3

3.5 Task 5: Analysis of IMDb Scores by Number of Votes

3.5.1 Objective

The objective of this task is to compute the number of movies and the average IMDb score categorized by bins of the number of IMDb votes. Meaningful bins will be created to categorize IMDb votes into ranges.

3.5.2 SQL Query

```
SELECT  CASE
        WHEN imdb_votes <= 1000 THEN '0-1000'
        WHEN imdb_votes > 1000 AND imdb_votes <= 10000 THEN '1001-10000
            '
        WHEN imdb_votes > 10000 AND imdb_votes <= 40000 THEN '
            10001-40000'
        WHEN imdb_votes > 40000 AND imdb_votes <= 70000 THEN '
            40001-70000'
        WHEN imdb_votes > 70000 THEN '>70000'
        ELSE 'Unknown'
    END AS vote_range,
    COUNT(*) AS num_movies,
    AVG(imdb_score) AS avg_imdb_score
FROM    deduplicated_list
GROUP BY    vote_range
ORDER BY    vote_range;
```

3.5.3 Results

Figure 2 provides insights into the relationship between the number of IMDb votes a movie receives and its average IMDb score.

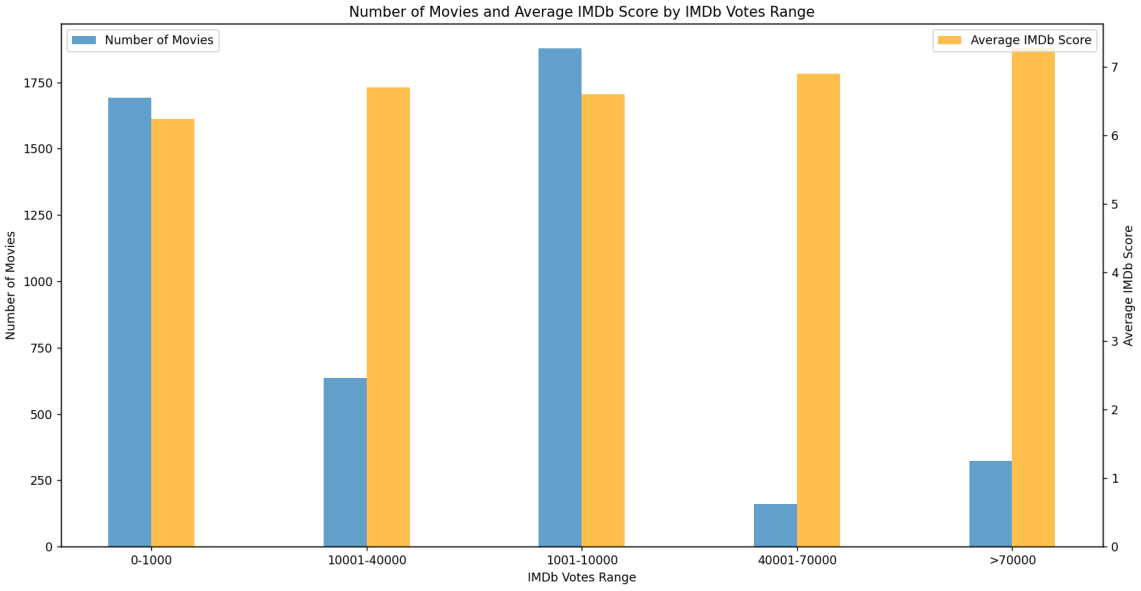

Figure 2: Number of Movies and Average IMDb Score by IMDb Votes Range

3.5.4 Interpretation

- **Number of Movies (Blue Bars)**:
  - The highest number of movies fall into the 0-10,000 votes range.
  - As the number of votes increases, the number of movies in each category decreases significantly.
- **Average IMDb Score (Orange Line)**:
  - Movies with 0-10,000 votes have an average IMDb score just below 6.
  - As the number of votes increases, the average IMDb score also increases.
  - Movies with more than 100,000 votes have the highest average IMDb score, just over 7.

3.6 Task 6: Number of Movies per Genre

3.6.1 Objective

The objective of this task is to count the number of movies per genre and visualize the distribution of movie genres.

3.6.2 SQL Query

```
SELECT genres, COUNT(*) AS movie_count
FROM titles
WHERE "type" = 'MOVIE'
GROUP BY genres
ORDER BY movie_count DESC;
```

3.6.3 Results

Figure 3 provides insights into the distribution of movies across different genres. Due to the extensive data, the visualization focuses on the top 10 and bottom 10 genres:

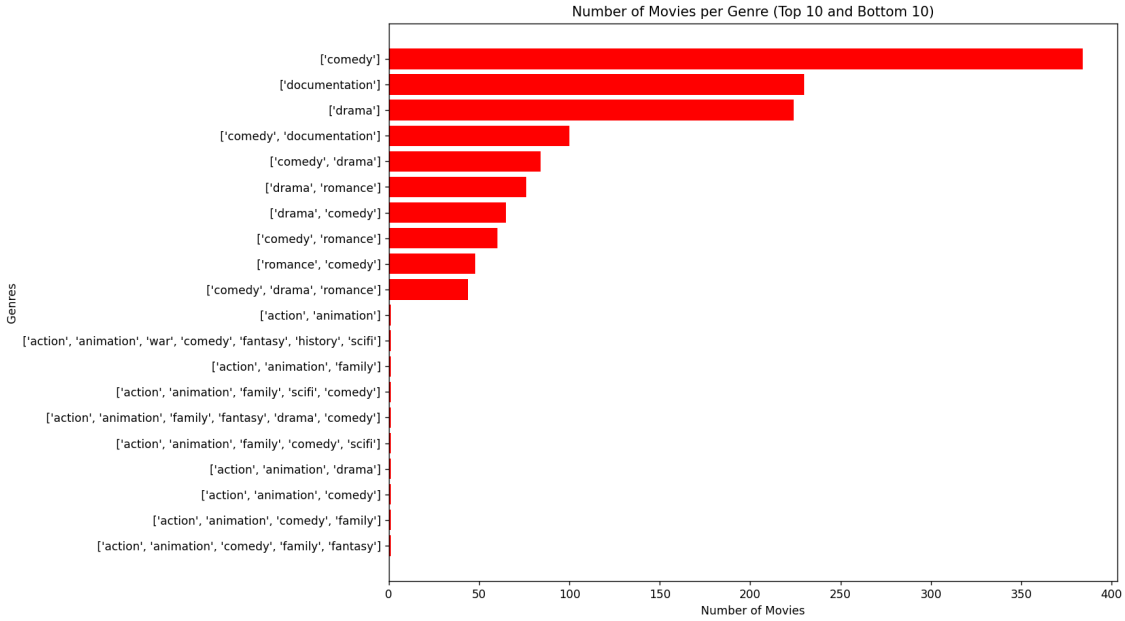

Figure 3: Number of Movies per Genre

3.6.4 Interpretation

- **Comedy** is the most common genre, indicating its broad appeal and versatility.
- **Drama** also appears frequently, both as a standalone genre and in combination with others.
- Genres in the bottom 10 often involve multiple genre combinations, suggesting these specific mixes are less common in the film industry.

This visualization helps understand the popularity and prevalence of different movie genres.

3.7 Task 7: Actor with Most Movie Appearances by Release Year

3.7.1 Objective

Determine the actor who starred in the most movies each year by release year and quantify the number of movies they starred in during their busiest year.

3.7.2 SQL Query

```
WITH ActorMovieCounts AS (
    SELECT  dl.release_year, c.name AS actor, COUNT(*) AS movie_count,
        ROW_NUMBER() OVER (PARTITION BY dl.release_year ORDER BY COUNT(
           *) DESC) AS row_num
    FROM    deduplicated_list dl
    JOIN    credits c ON dl.id = c.id
    WHERE   c.role = 'ACTOR'
    GROUP BY    dl.release_year, c.name
)
SELECT release_year, actor, movie_count
FROM    ActorMovieCounts
WHERE   row_num = 1
ORDER BY    movie_count DESC;
```

1. **Common Table Expression (CTE) - ActorMovieCounts**:
    - Selects the release year, actor name (`c.name`), and counts the number of movies (`COUNT(*)`) each actor starred in.
    - Uses the `ROW_NUMBER()` function partitioned by `dl.release_year` to rank actors by the number of movies they starred in, in descending order (`ORDER BY COUNT(*) DESC`).
    - Filters actors who have the role of 'ACTOR' (`c.role = 'ACTOR'`).
2. **Main Query**:
    - Selects the release year, actor name, and movie count from the CTE (`ActorMovieCounts`).
    - Filters rows where `row_num = 1`, selecting only the top actor by movie count for each release year.
    - Orders the result by movie count in descending order.

3.7.3 Results

The Table 3 and Figure 4 shows that Yuki Kaji and Tiffany Haddish are the actors that are starred most in movies.

Release Year	Actor	Movies Starred
2018	Yuki Kaji	6
2019	Tiffany Haddish	6
2009	Kareena Kapoor Khan	5
2021	Kari Wahlgren	5
2005	Nada Abou Farhat	4
.	.	.
.	.	.
1990	Yusef Bulos	1
1991	William Devane	1
1992	William Hootkins	1
1996	William Allen Young	1
1992	William Hootkins	1

Table 3: Actors with Most Movie Appearances

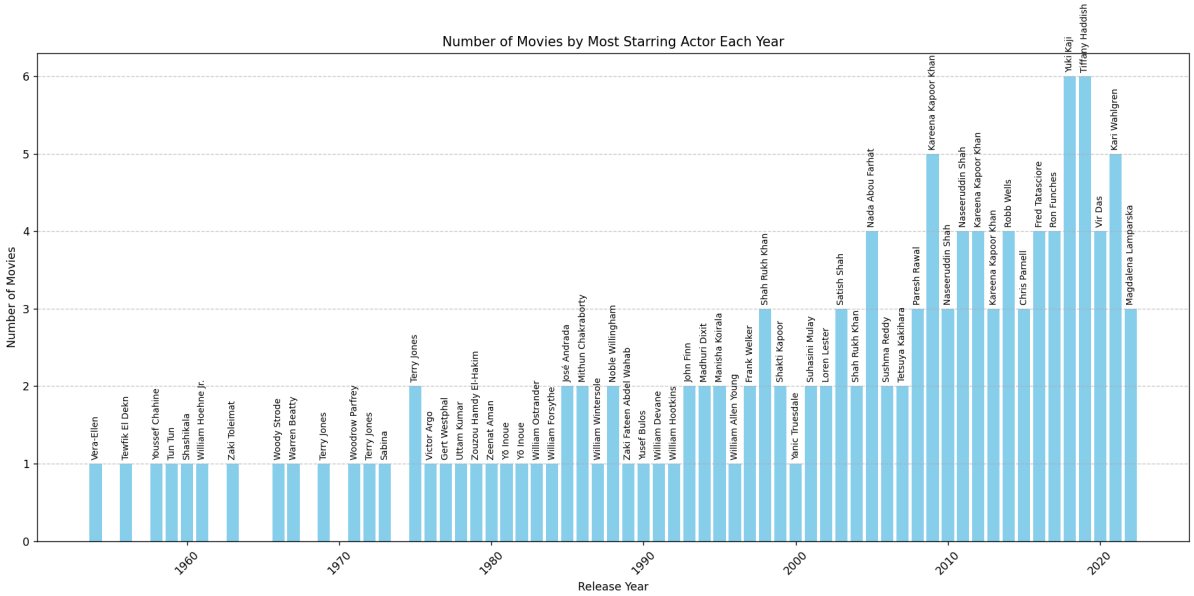

Figure 4: Number of Movies by Most Starring Actor each year

3.8 Task 8: Number of Movies w.r.t Year

3.8.1 Objective

Determine the number of movies with respect to year, digging out the year with the most number of movies.

3.8.2 SQL Query

```
WITH MovieCounts AS (
    SELECT  release_year, COUNT(*) AS num_movies
    FROM    deduplicated_list
    WHERE   type = 'MOVIE'  -- Consider only movies
    GROUP BY    release_year
)
SELECT *
FROM    MovieCounts
```

3.8.3 Results

Table 4 and Figure 5 shows the number of movies for each year, with 2019 being the year with the most number of movies.

Release Year	Number of Movies
1954	1
1956	1
1958	1
1959	1
1960	1
.	.
.	.
2018	405
2019	425
2020	399
2021	400
2022	89

Table 4: Number of Movies w.r.t Year

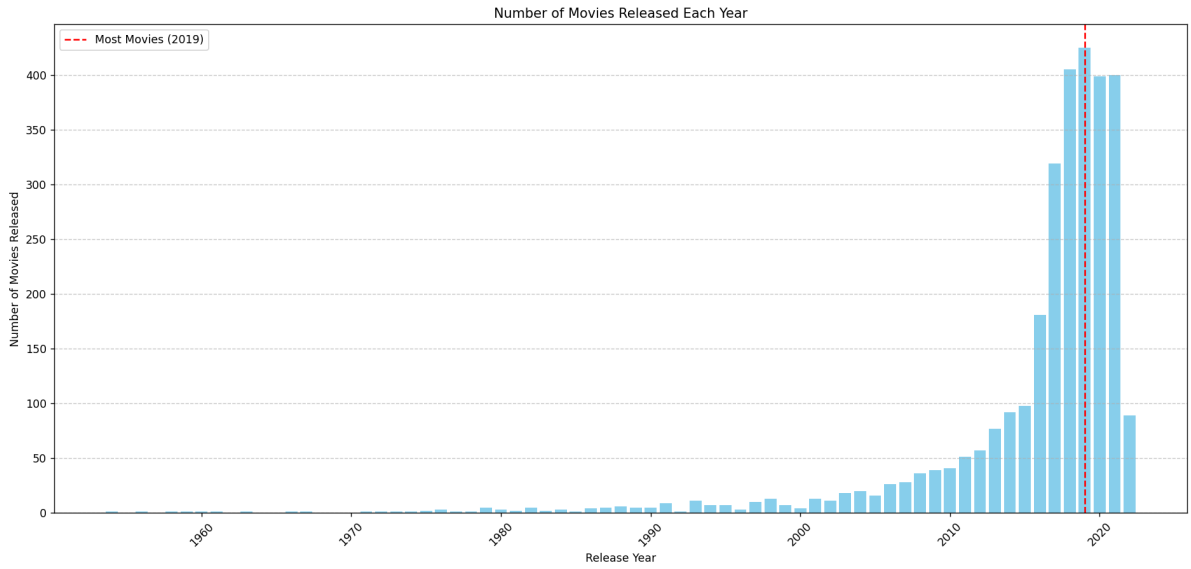

Figure 5: Number of Movies w.r.t Year

4 Conclusion

In conclusion, this project successfully showed how to connect to a database, use SQL to import and modify data, and create useful visualizations. By performing tasks like trend analysis, data cleaning, and examining genre distribution, we uncovered key patterns and insights in a large movie dataset. The SQL queries and visualizations demonstrated strong skills in database management and data analysis, offering a clear and effective way to explore the data. This report highlights the valuable insights that can come from well-organized data, underscoring the importance of data-driven decisions in product analysis.