# Advanced Database Management and Query Optimization for a Movie Dataset

*Phase 2*

Rohan Patel
ESDS
University At Buffalo
Buffalo, NY
rpatel38@buffalo.edu

Jay Thanki
ESDS
University At Buffalo
Buffalo, NY
jayyoges@buffalo.edu

*Abstract*— **This detailed report delves into the management and optimization of a complex movie database derived from various sources such as Movies.csv, Links.csv, Tags.csv, Ratings.csv, and ml-youtube.csv. The research highlights the importance of data integrity, efficient query execution, and database normalization. It provides a case study of the difficulties faced and creative solutions used while managing enormous datasets in a PostgreSQL environment.**

**Keywords— SQL, ER Model, Relational Model, MovieLens Data, Data Administration, Query Optimization, Deployment.**

## I. Handling Large Datasets (Task 5)

We had a number of noteworthy difficulties in handling this large dataset, especially in the data loading stage. The CSV files' various formats were the main source of the difficulty. In particular, several files applied the field terminators—double quotes—inconsistently in various rows and columns. To guarantee effective data integration, distinct data loading syntax had to be created for every file due to the formatting discrepancy.

The Ratings file, which contained more than two million entries, presented another well-known issue. Long loading times into the temporary table—a crucial stage in our data processing pipeline—were caused by the sheer volume of this data. This file's size not only made loading take longer, but it also made it take longer to change the data types in these columns before integrating them into the main table that is always open.

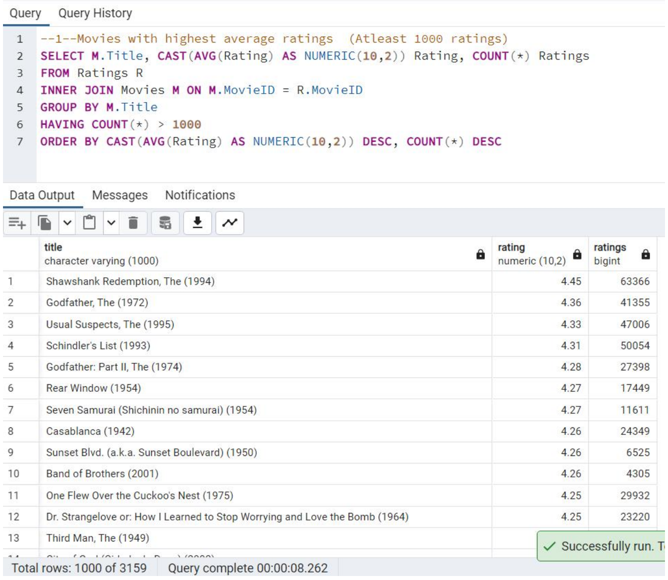
We used temporary tables, which run in memory and hence provide faster processing speeds than the main table, to speed up the data loading process. But indexing, which is usually a powerful strategy for handling enormous datasets, was considered impractical in this case. The choice was made based on the fact that these tables were only meant to be temporary; indexing them would have required an unnecessary use of resources and space because they were going to be deleted after the data was cleaned and loaded.

## II. SQL Query Testing and Execution (Task 6)

We have tested our database using a number of SQL queries in this project section. A broad range of actions, such as insertion, deletion, updating, and intricate selection procedures, were included in the design of these tests. An extensive summary of the queries utilized and their particular goals is given below:

### 1. Identifying Movies with Highest Average Ratings

- **Query:** The movies with the highest average ratings were found using a SELECT statement, with a minimum of 1000 ratings required for a movie to be considered. In order to organize the Movies and Ratings tables according to movie titles, an inner join was performed.
- **Purpose:** To demonstrate how aggregate functions with GROUP BY and HAVING clauses can yield useful information, like a list of the highest rated films in our dataset.



```
--1--Movies with highest average ratings  (Atleast 1000 ratings)
SELECT M.Title, CAST(AVG(Rating) AS NUMERIC(10,2)) Rating, COUNT(*) Ratings
FROM Ratings R
INNER JOIN Movies M ON M.MovieID = R.MovieID
GROUP BY M.Title
HAVING COUNT(*) > 1000
ORDER BY CAST(AVG(Rating) AS NUMERIC(10,2)) DESC, COUNT(*) DESC
```

| | title<br>character varying (1000) | rating<br>numeric (10,2) | ratings<br>bigint |
|---|---|---|---|
| 1 | Shawshank Redemption, The (1994) | 4.45 | 63366 |
| 2 | Godfather, The (1972) | 4.36 | 41355 |
| 3 | Usual Suspects, The (1995) | 4.33 | 47006 |
| 4 | Schindler's List (1993) | 4.31 | 50054 |
| 5 | Godfather: Part II, The (1974) | 4.28 | 27398 |
| 6 | Rear Window (1954) | 4.27 | 17449 |
| 7 | Seven Samurai (Shichinin no samurai) (1954) | 4.27 | 11611 |
| 8 | Casablanca (1942) | 4.26 | 24349 |
| 9 | Sunset Blvd. (a.k.a. Sunset Boulevard) (1950) | 4.26 | 6525 |
| 10 | Band of Brothers (2001) | 4.26 | 4305 |
| 11 | One Flew Over the Cuckoo's Nest (1975) | 4.25 | 29932 |
| 12 | Dr. Strangelove or: How I Learned to Stop Worrying and Love the Bomb (1964) | 4.25 | 23220 |
| 13 | Third Man, The (1949) | | |

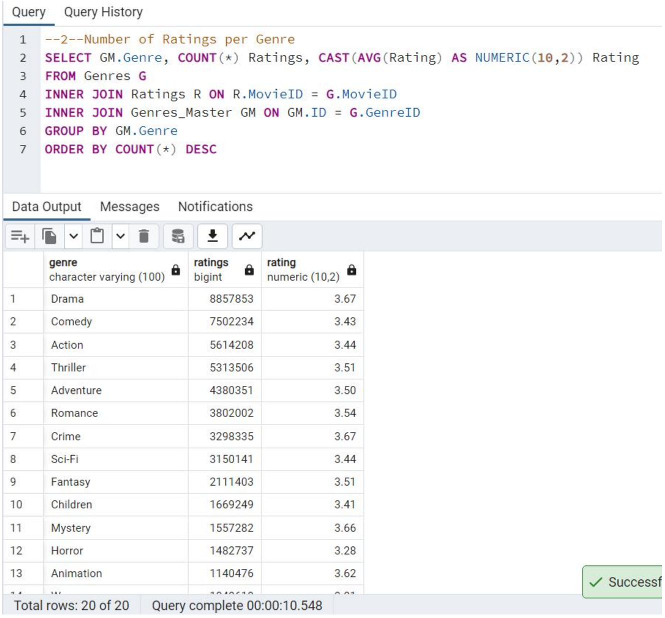
✓ Successfully run. T

Total rows: 1000 of 3159    Query complete 00:00:08.262

Fig. 1. Identifying Movies with Highest Average Ratings

### 2. Calculating Number of Ratings per Genre

- **Query:** To determine the total number of ratings and average rating per genre, a sophisticated SELECT statement linking the Genres, Ratings, and Genres_ Master databases was required.
- **Purpose:** to illustrate the capacity to aggregate data across various dimensions—in this case, genres—and link multiple tables.



```
--2--Number of Ratings per Genre
SELECT GM.Genre, COUNT(*) Ratings, CAST(AVG(Rating) AS NUMERIC(10,2)) Rating
FROM Genres G
INNER JOIN Ratings R ON R.MovieID = G.MovieID
INNER JOIN Genres_Master GM ON GM.ID = G.GenreID
GROUP BY GM.Genre
ORDER BY COUNT(*) DESC
```

| | genre<br>character varying (100) | ratings<br>bigint | rating<br>numeric (10,2) |
|---|---|---|---|
| 1 | Drama | 8857853 | 3.67 |
| 2 | Comedy | 7502234 | 3.43 |
| 3 | Action | 5614208 | 3.44 |
| 4 | Thriller | 5313506 | 3.51 |
| 5 | Adventure | 4380351 | 3.50 |
| 6 | Romance | 3802002 | 3.54 |
| 7 | Crime | 3298335 | 3.67 |
| 8 | Sci-Fi | 3150141 | 3.44 |
| 9 | Fantasy | 2111403 | 3.51 |
| 10 | Children | 1669249 | 3.41 |
| 11 | Mystery | 1557282 | 3.66 |
| 12 | Horror | 1482737 | 3.28 |
| 13 | Animation | 1140476 | 3.62 |

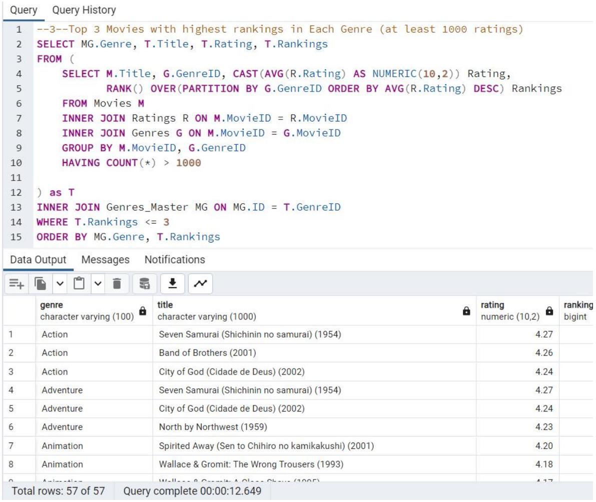
✓ Successf

Total rows: 20 of 20    Query complete 00:00:10.548

Fig. 2. Calculating Number of Ratings per Genre

### 3. Top Movies in Each Genre

- **Query:** This query, which required that the films have at least 1000 ratings, retrieved the top 3 films with the highest rankings in each category by using subqueries and window functions.
- **Purpose:** To demonstrate how to use window functions (RANK() OVER()) and subqueries, two advanced SQL features, for more in-depth data analysis.



```sql
--3--Top 3 Movies with highest rankings in Each Genre (at least 1000 ratings)
SELECT MG.Genre, T.Title, T.Rating, T.Rankings
FROM (
    SELECT M.Title, G.GenreID, CAST(AVG(R.Rating) AS NUMERIC(10,2)) Rating,
           RANK() OVER(PARTITION BY G.GenreID ORDER BY AVG(R.Rating) DESC) Rankings
    FROM Movies M
    INNER JOIN Ratings R ON M.MovieID = R.MovieID
    INNER JOIN Genres G ON M.MovieID = G.MovieID
    GROUP BY M.MovieID, G.GenreID
    HAVING COUNT(*) > 1000

) as T
INNER JOIN Genres_Master MG ON MG.ID = T.GenreID
WHERE T.Rankings <= 3
ORDER BY MG.Genre, T.Rankings
```

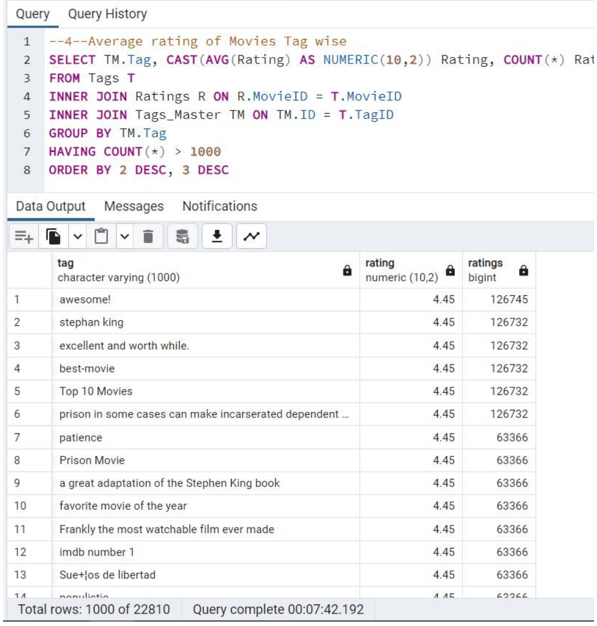
| | genre<br>character varying (100) | title<br>character varying (1000) | rating<br>numeric (10,2) | ranking<br>bigint |
|---|---|---|---|---|
| 1 | Action | Seven Samurai (Shichinin no samurai) (1954) | 4.27 | |
| 2 | Action | Band of Brothers (2001) | 4.26 | |
| 3 | Action | City of God (Cidade de Deus) (2002) | 4.24 | |
| 4 | Adventure | Seven Samurai (Shichinin no samurai) (1954) | 4.27 | |
| 5 | Adventure | City of God (Cidade de Deus) (2002) | 4.24 | |
| 6 | Adventure | North by Northwest (1959) | 4.23 | |
| 7 | Animation | Spirited Away (Sen to Chihiro no kamikakushi) (2001) | 4.20 | |
| 8 | Animation | Wallace & Gromit: The Wrong Trousers (1993) | 4.18 | |
| 9 | Animation | Wallace & Gromit: A Close Shave (1995) | 4.17 | |

Total rows: 57 of 57 Query complete 00:00:12.649

Fig. 3 Top Movies in Each Genre

### 4. Average Movie Rating by Tag

- **Query:** The average rating for movies, sorted by tags, was determined by doing an INNER JOIN across the Tags, Ratings, and Tags_Master columns.
- **Purpose:** to provide an example of how to carry out intricate data aggregations across various categories—in this case, movie tags.



```sql
--4--Average rating of Movies Tag wise
SELECT TM.Tag, CAST(AVG(Rating) AS NUMERIC(10,2)) Rating, COUNT(*) Rat
FROM Tags T
INNER JOIN Ratings R ON R.MovieID = T.MovieID
INNER JOIN Tags_Master TM ON TM.ID = T.TagID
GROUP BY TM.Tag
HAVING COUNT(*) > 1000
ORDER BY 2 DESC, 3 DESC
```

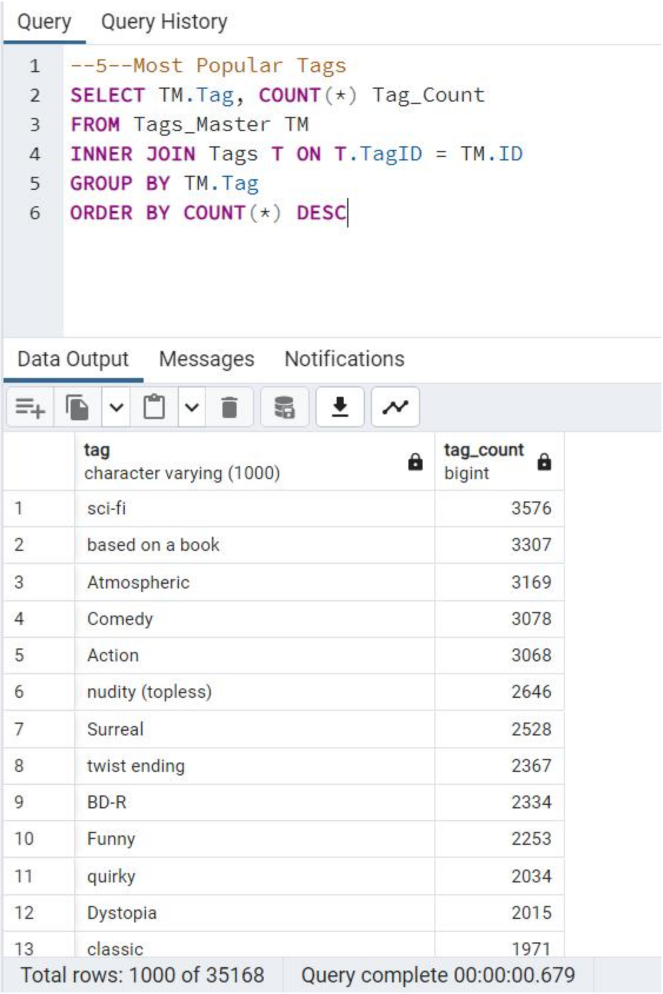
| | tag<br>character varying (1000) | rating<br>numeric (10,2) | ratings<br>bigint |
|---|---|---|---|
| 1 | awesome! | 4.45 | 126745 |
| 2 | stephan king | 4.45 | 126732 |
| 3 | excellent and worth while. | 4.45 | 126732 |
| 4 | best-movie | 4.45 | 126732 |
| 5 | Top 10 Movies | 4.45 | 126732 |
| 6 | prison in some cases can make incarserated dependent ... | 4.45 | 126732 |
| 7 | patience | 4.45 | 63366 |
| 8 | Prison Movie | 4.45 | 63366 |
| 9 | a great adaptation of the Stephen King book | 4.45 | 63366 |
| 10 | favorite movie of the year | 4.45 | 63366 |
| 11 | Frankly the most watchable film ever made | 4.45 | 63366 |
| 12 | imdb number 1 | 4.45 | 63366 |
| 13 | Sue+¦os de libertad | 4.45 | 63366 |
| 14 | populistic | 4.45 | 63366 |

Total rows: 1000 of 22810 Query complete 00:07:42.192

Fig. 4 Average Movie Rating by Tag

### 5. Identifying Most Popular Tags

- **Query:** To determine each tag's popularity, a straightforward yet powerful SELECT statement was utilized to count the occurrences of each one.
- **Purpose:** to emphasize how popular movie tags are determined by using the GROUP BY and ORDER BY clauses.



```sql
--5--Most Popular Tags
SELECT TM.Tag, COUNT(*) Tag_Count
FROM Tags_Master TM
INNER JOIN Tags T ON T.TagID = TM.ID
GROUP BY TM.Tag
ORDER BY COUNT(*) DESC
```

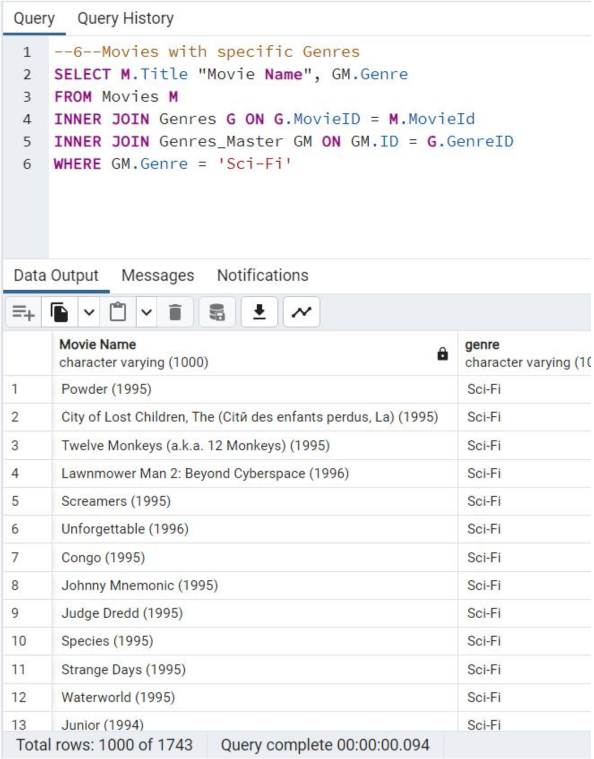
| | tag<br>character varying (1000) | tag_count<br>bigint |
|---|---|---|
| 1 | sci-fi | 3576 |
| 2 | based on a book | 3307 |
| 3 | Atmospheric | 3169 |
| 4 | Comedy | 3078 |
| 5 | Action | 3068 |
| 6 | nudity (topless) | 2646 |
| 7 | Surreal | 2528 |
| 8 | twist ending | 2367 |
| 9 | BD-R | 2334 |
| 10 | Funny | 2253 |
| 11 | quirky | 2034 |
| 12 | Dystopia | 2015 |
| 13 | classic | 1971 |

Total rows: 1000 of 35168 Query complete 00:00:00.679

Fig. 5 Top Movies in Each Genre

### 6. & 7. Movies Filtered by Specific Genres and Tags

- **Queries:** These SELECT lines demonstrate the database's capacity to manage simple data retrieval based on predetermined criteria by filtering movies by particular genres and tags.

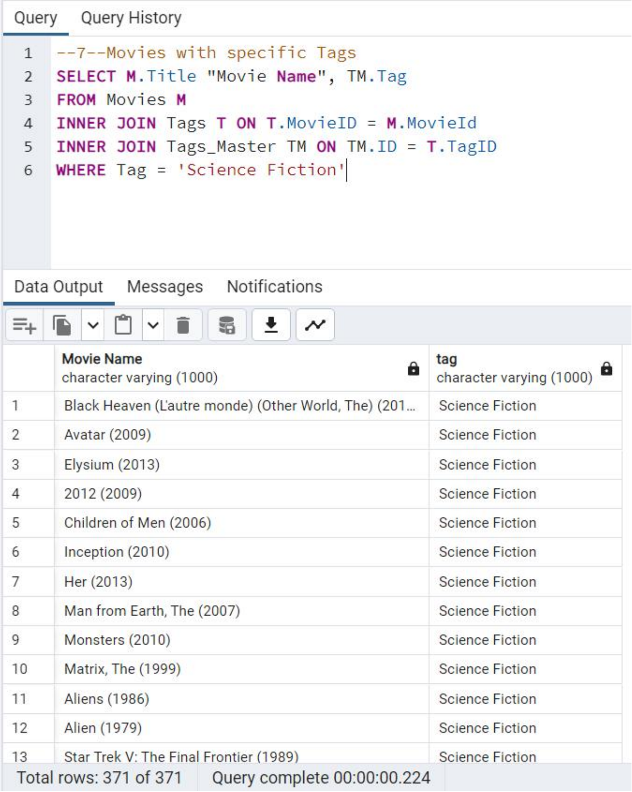


```
--6--Movies with specific Genres
SELECT M.Title "Movie Name", GM.Genre
FROM Movies M
INNER JOIN Genres G ON G.MovieID = M.MovieId
INNER JOIN Genres_Master GM ON GM.ID = G.GenreID
WHERE GM.Genre = 'Sci-Fi'
```

| | Movie Name<br>character varying (1000) | genre<br>character varying (1( |
|---|---|---|
| 1 | Powder (1995) | Sci-Fi |
| 2 | City of Lost Children, The (Citй des enfants perdus, La) (1995) | Sci-Fi |
| 3 | Twelve Monkeys (a.k.a. 12 Monkeys) (1995) | Sci-Fi |
| 4 | Lawnmower Man 2: Beyond Cyberspace (1996) | Sci-Fi |
| 5 | Screamers (1995) | Sci-Fi |
| 6 | Unforgettable (1996) | Sci-Fi |
| 7 | Congo (1995) | Sci-Fi |
| 8 | Johnny Mnemonic (1995) | Sci-Fi |
| 9 | Judge Dredd (1995) | Sci-Fi |
| 10 | Species (1995) | Sci-Fi |
| 11 | Strange Days (1995) | Sci-Fi |
| 12 | Waterworld (1995) | Sci-Fi |
| 13 | Junior (1994) | Sci-Fi |

Total rows: 1000 of 1743    Query complete 00:00:00.094

Fig. 6 Movies Filtered by Specific Genre



```
--7--Movies with specific Tags
SELECT M.Title "Movie Name", TM.Tag
FROM Movies M
INNER JOIN Tags T ON T.MovieID = M.MovieId
INNER JOIN Tags_Master TM ON TM.ID = T.TagID
WHERE Tag = 'Science Fiction'
```

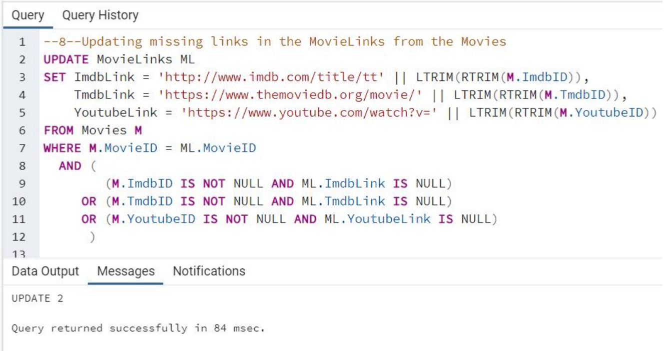
| | Movie Name<br>character varying (1000) | tag<br>character varying (1000) |
|---|---|---|
| 1 | Black Heaven (L’autre monde) (Other World, The) (201... | Science Fiction |
| 2 | Avatar (2009) | Science Fiction |
| 3 | Elysium (2013) | Science Fiction |
| 4 | 2012 (2009) | Science Fiction |
| 5 | Children of Men (2006) | Science Fiction |
| 6 | Inception (2010) | Science Fiction |
| 7 | Her (2013) | Science Fiction |
| 8 | Man from Earth, The (2007) | Science Fiction |
| 9 | Monsters (2010) | Science Fiction |
| 10 | Matrix, The (1999) | Science Fiction |
| 11 | Aliens (1986) | Science Fiction |
| 12 | Alien (1979) | Science Fiction |
| 13 | Star Trek V: The Final Frontier (1989) | Science Fiction |

Total rows: 371 of 371    Query complete 00:00:00.224

Fig. 7 Movies Filtered by Specific Tag

### 8. Updating Missing Links in MovieLinks

- **Query:** A UPDATE statement was used to add missing YouTube, TMDb, and IMDb links in the MovieLinks table using matching data from the Movies table.
- **Purpose:** to demonstrate the use of UPDATE procedures to synchronize data across linked tables.



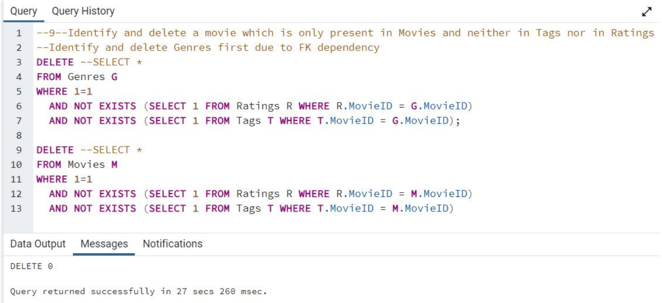
```
--8--Updating missing links in the MovieLinks from the Movies
UPDATE MovieLinks ML
SET ImdbLink = 'http://www.imdb.com/title/tt' || LTRIM(RTRIM(M.ImdbID)),
    TmdbLink = 'https://www.themoviedb.org/movie/' || LTRIM(RTRIM(M.TmdbID)),
    YoutubeLink = 'https://www.youtube.com/watch?v=' || LTRIM(RTRIM(M.YoutubeID))
FROM Movies M
WHERE M.MovieID = ML.MovieID
  AND (
        (M.ImdbID IS NOT NULL AND ML.ImdbLink IS NULL)
     OR (M.TmdbID IS NOT NULL AND ML.TmdbLink IS NULL)
     OR (M.YoutubeID IS NOT NULL AND ML.YoutubeLink IS NULL)
      )
```

```
UPDATE 2

Query returned successfully in 84 msec.
```

Fig. 8 Updating Missing Links in MovieLinks

### 9. Deletion of Redundant Data

- **Query:** The process entailed eliminating films and categories that weren't mentioned in the Ratings or Tags tables.
- **Purpose:** to guarantee that there are no orphaned records in the database and to preserve data integrity.



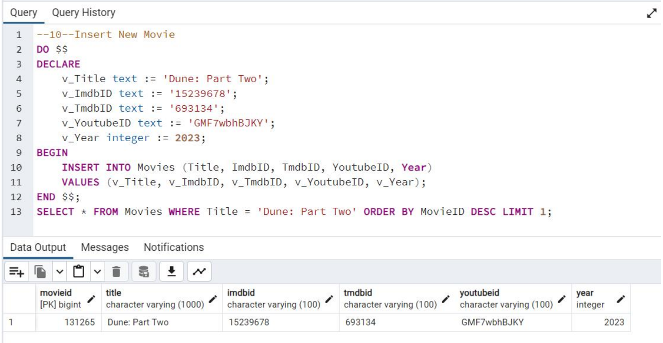
```
--9--Identify and delete a movie which is only present in Movies and neither in Tags nor in Ratings
--Identify and delete Genres first due to FK dependency
DELETE --SELECT *
FROM Genres G
WHERE 1=1
  AND NOT EXISTS (SELECT 1 FROM Ratings R WHERE R.MovieID = G.MovieID)
  AND NOT EXISTS (SELECT 1 FROM Tags T WHERE T.MovieID = G.MovieID);

DELETE --SELECT *
FROM Movies M
WHERE 1=1
  AND NOT EXISTS (SELECT 1 FROM Ratings R WHERE R.MovieID = M.MovieID)
  AND NOT EXISTS (SELECT 1 FROM Tags T WHERE T.MovieID = M.MovieID)
```

```
DELETE 0

Query returned successfully in 27 secs 260 msec.
```

Fig. 9 Deletion of Redundant Data

### 10. Inserting a New Movie and Associated Links

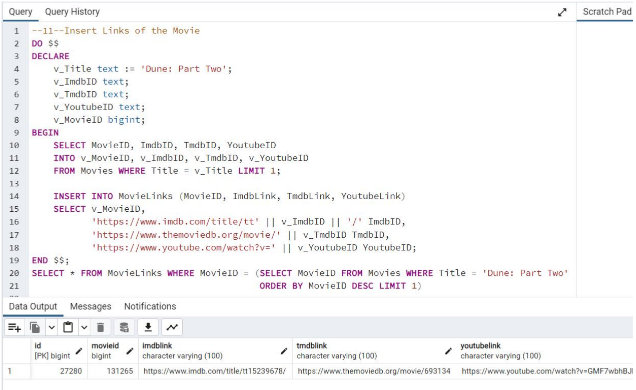
- **Query:** We added 'Dune: Part Two' as a new movie and its associated links to the database by using the DO command.
- **Purpose:** to demonstrate the insertion procedure and the connections between various tables in our database.



```
--10--Insert New Movie
DO $$
DECLARE
    v_Title text := 'Dune: Part Two';
    v_ImdbID text := '15239678';
    v_TmdbID text := '693134';
    v_YoutubeID text := 'GMF7wbhBJKY';
    v_Year integer := 2023;
BEGIN
    INSERT INTO Movies (Title, ImdbID, TmdbID, YoutubeID, Year)
    VALUES (v_Title, v_ImdbID, v_TmdbID, v_YoutubeID, v_Year);
END $$;
SELECT * FROM Movies WHERE Title = 'Dune: Part Two' ORDER BY MovieID DESC LIMIT 1;
```

| | movieid<br>[PK] bigint | title<br>character varying (1000) | imdbid<br>character varying (100) | tmdbid<br>character varying (100) | youtubeid<br>character varying (100) | year<br>integer |
|---|---|---|---|---|---|---|
| 1 | 131265 | Dune: Part Two | 15239678 | 693134 | GMF7wbhBJKY | 2023 |

Fig. 10 Insert New Movie



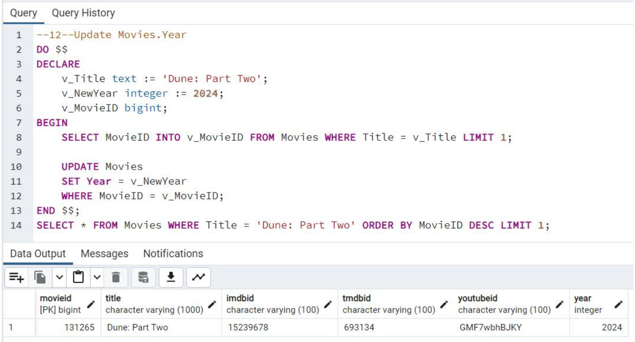
```sql
--11--Insert Links of the Movie
DO $$
DECLARE
    v_Title text := 'Dune: Part Two';
    v_ImdbID text;
    v_TmdbID text;
    v_YoutubeID text;
    v_MovieID bigint;
BEGIN
    SELECT MovieID, ImdbID, TmdbID, YoutubeID
    INTO v_MovieID, v_ImdbID, v_TmdbID, v_YoutubeID
    FROM Movies WHERE Title = v_Title LIMIT 1;

    INSERT INTO MovieLinks (MovieID, ImdbLink, TmdbLink, YoutubeLink)
    SELECT v_MovieID,
           'https://www.imdb.com/title/tt' || v_ImdbID || '/' ImdbID,
           'https://www.themoviedb.org/movie/' || v_TmdbID TmdbID,
           'https://www.youtube.com/watch?v=' || v_YoutubeID YoutubeID;
END $$;
SELECT * FROM MovieLinks WHERE MovieID = (SELECT MovieID FROM Movies WHERE Title = 'Dune: Part Two'
                                          ORDER BY MovieID DESC LIMIT 1)
```

| | id<br>[PK] bigint | movieid<br>bigint | imdblink<br>character varying (100) | tmdblink<br>character varying (100) | youtubelink<br>character varying (100) |
|---|---|---|---|---|---|
| 1 | 27280 | 131265 | https://www.imdb.com/title/tt15239678/ | https://www.themoviedb.org/movie/693134 | https://www.youtube.com/watch?v=GMF7wbhBJ |

Fig. 11 Insert Links of the Movie

## 11. Updating and Deleting Movie Data

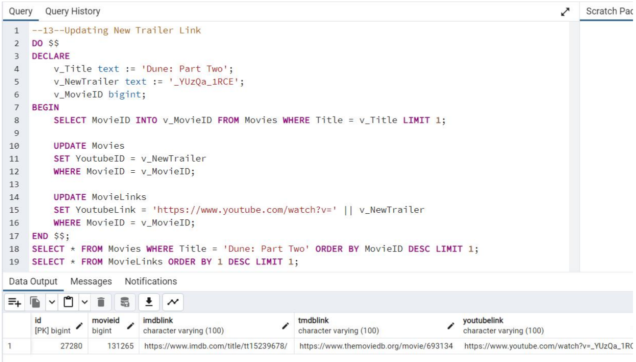
- **Query:** These queries concerned the removal of movie links and individual movies, as well as the upgrading of particular movie details (such as a new trailer link).
- **Purpose:** To illustrate the dynamic nature of the database in handling updates and deletions, reflecting real-world changes.



```sql
--12--Update Movies.Year
DO $$
DECLARE
    v_Title text := 'Dune: Part Two';
    v_NewYear integer := 2024;
    v_MovieID bigint;
BEGIN
    SELECT MovieID INTO v_MovieID FROM Movies WHERE Title = v_Title LIMIT 1;

    UPDATE Movies
    SET Year = v_NewYear
    WHERE MovieID = v_MovieID;
END $$;
SELECT * FROM Movies WHERE Title = 'Dune: Part Two' ORDER BY MovieID DESC LIMIT 1;
```

| | movieid<br>[PK] bigint | title<br>character varying (1000) | imdbid<br>character varying (100) | tmdbid<br>character varying (100) | youtubeid<br>character varying (100) | year<br>integer |
|---|---|---|---|---|---|---|
| 1 | 131265 | Dune: Part Two | 15239678 | 693134 | GMF7wbhBJKY | 2024 |

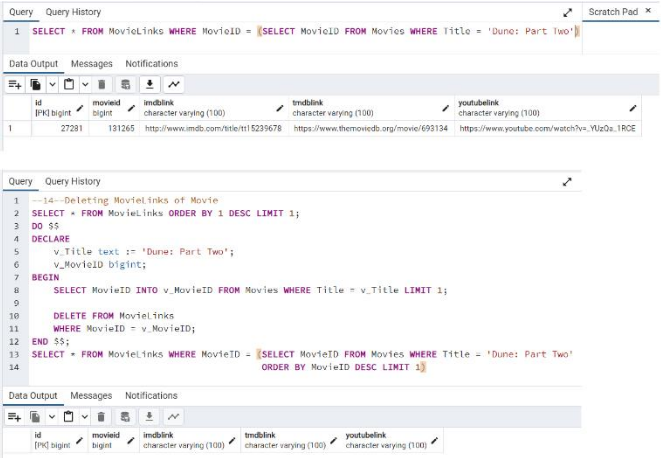
Fig. 12 Update Movies. Year



```sql
--13--Updating New Trailer Link
DO $$
DECLARE
    v_Title text := 'Dune: Part Two';
    v_NewTrailer text := '_YUzQa_1RCE';
    v_MovieID bigint;
BEGIN
    SELECT MovieID INTO v_MovieID FROM Movies WHERE Title = v_Title LIMIT 1;

    UPDATE Movies
    SET YoutubeID = v_NewTrailer
    WHERE MovieID = v_MovieID;

    UPDATE MovieLinks
    SET YoutubeLink = 'https://www.youtube.com/watch?v=' || v_NewTrailer
    WHERE MovieID = v_MovieID;
END $$;
SELECT * FROM Movies WHERE Title = 'Dune: Part Two' ORDER BY MovieID DESC LIMIT 1;
SELECT * FROM MovieLinks ORDER BY 1 DESC LIMIT 1;
```

| | id<br>[PK] bigint | movieid<br>bigint | imdblink<br>character varying (100) | tmdblink<br>character varying (100) | youtubelink<br>character varying (100) |
|---|---|---|---|---|---|
| 1 | 27280 | 131265 | https://www.imdb.com/title/tt15239678/ | https://www.themoviedb.org/movie/693134 | https://www.youtube.com/watch?v=_YUzQa_1RC |

Fig. 13 Updating New Trailer Link



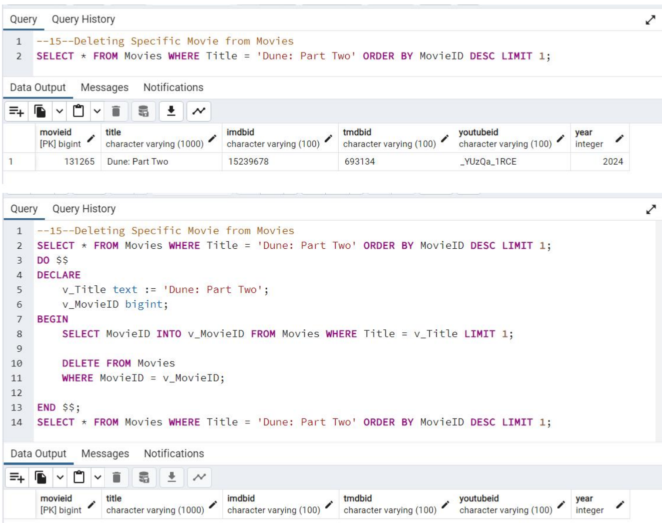
```sql
SELECT * FROM MovieLinks WHERE MovieID = (SELECT MovieID FROM Movies WHERE Title = 'Dune: Part Two')
```

| | id<br>[PK] bigint | movieid<br>bigint | imdblink<br>character varying (100) | tmdblink<br>character varying (100) | youtubelink<br>character varying (100) |
|---|---|---|---|---|---|
| 1 | 27281 | 131265 | http://www.imdb.com/title/tt15239678 | https://www.themoviedb.org/movie/693134 | https://www.youtube.com/watch?v=_YUzQa_1RCE |

```sql
--14--Deleting MovieLinks of Movie
SELECT * FROM MovieLinks ORDER BY 1 DESC LIMIT 1;
DO $$
DECLARE
    v_Title text := 'Dune: Part Two';
    v_MovieID bigint;
BEGIN
    SELECT MovieID INTO v_MovieID FROM Movies WHERE Title = v_Title LIMIT 1;

    DELETE FROM MovieLinks
    WHERE MovieID = v_MovieID;
END $$;
SELECT * FROM MovieLinks WHERE MovieID = (SELECT MovieID FROM Movies WHERE Title = 'Dune: Part Two'
                                          ORDER BY MovieID DESC LIMIT 1)
```

| | id<br>[PK] bigint | movieid<br>bigint | imdblink<br>character varying (100) | tmdblink<br>character varying (100) | youtubelink<br>character varying (100) |
|---|---|---|---|---|---|

Fig. 14 Deleting MovieLinks of Movie



```sql
--15--Deleting Specific Movie from Movies
SELECT * FROM Movies WHERE Title = 'Dune: Part Two' ORDER BY MovieID DESC LIMIT 1;
```

| | movieid<br>[PK] bigint | title<br>character varying (1000) | imdbid<br>character varying (100) | tmdbid<br>character varying (100) | youtubeid<br>character varying (100) | year<br>integer |
|---|---|---|---|---|---|---|
| 1 | 131265 | Dune: Part Two | 15239678 | 693134 | _YUzQa_1RCE | 2024 |

```sql
--15--Deleting Specific Movie from Movies
SELECT * FROM Movies WHERE Title = 'Dune: Part Two' ORDER BY MovieID DESC LIMIT 1;
DO $$
DECLARE
    v_Title text := 'Dune: Part Two';
    v_MovieID bigint;
BEGIN
    SELECT MovieID INTO v_MovieID FROM Movies WHERE Title = v_Title LIMIT 1;

    DELETE FROM Movies
    WHERE MovieID = v_MovieID;

END $$;
SELECT * FROM Movies WHERE Title = 'Dune: Part Two' ORDER BY MovieID DESC LIMIT 1;
```

| | movieid<br>[PK] bigint | title<br>character varying (1000) | imdbid<br>character varying (100) | tmdbid<br>character varying (100) | youtubeid<br>character varying (100) | year<br>integer |
|---|---|---|---|---|---|---|

Fig. 15 Deleting Specific Movie from Movies

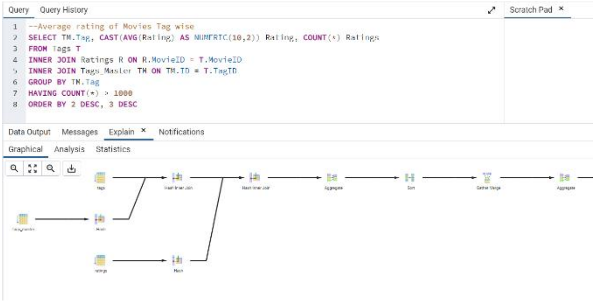
We ran each of these queries on our database and took screen grabs of the responses. These screenshots demonstrate the efficiency of the queries and the resilience of our database to handle different kinds of data operations. The variety of these queries, which range from straightforward data retrievals to intricate updates and aggregations, highlights the adaptability and effectiveness of our database system in handling and evaluating huge datasets.

## III. Query Execution Analysis (Task 7)

We performed a critical study of query execution throughout this project phase in order to pinpoint performance bottlenecks and apply fixes. We examined three troublesome queries using PostgreSQL's EXPLAIN tool, calculating their costs and coming up with ways to make them better.

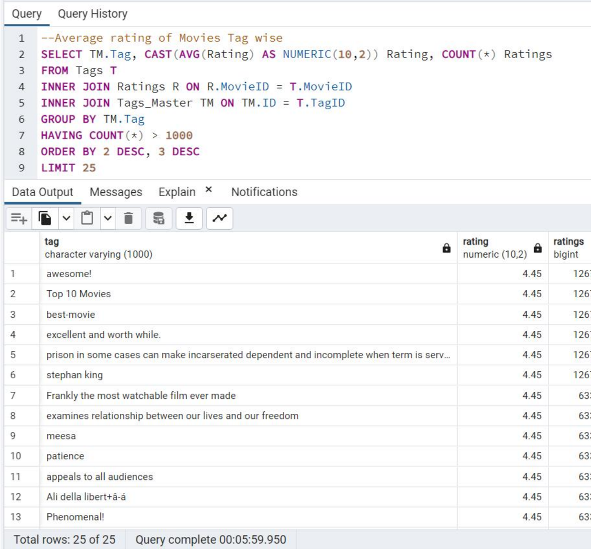
### 1. Average Rating of Movies by Tag

- **Initial Query and Issue:** The purpose of the query was to find the average movie rating by tag, but because of the quantity of the dataset, it was originally taking an unfeasible 6-7 minutes. The query was executing extremely inefficient sequential scans across three tables, as the EXPLAIN investigation showed.



```sql
--Average rating of Movies Tag wise
SELECT TM.Tag, CAST(AVG(Rating) AS NUMERIC(10,2)) Rating, COUNT(*) Ratings
FROM Tags T
INNER JOIN Ratings R ON R.MovieID = T.MovieID
INNER JOIN Tags_Master TM ON TM.ID = T.TagID
GROUP BY TM.Tag
HAVING COUNT(*) > 1000
ORDER BY 2 DESC, 3 DESC
```

Fig. 16. Explain tool result on query



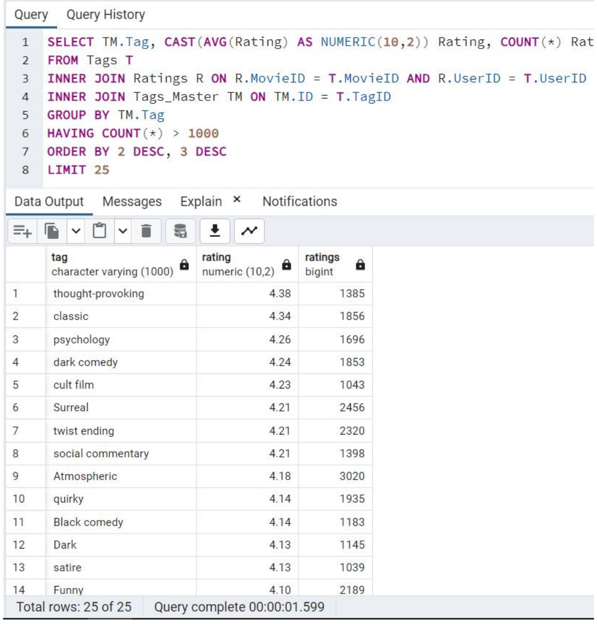
```sql
--Average rating of Movies Tag wise
SELECT TM.Tag, CAST(AVG(Rating) AS NUMERIC(10,2)) Rating, COUNT(*) Ratings
FROM Tags T
INNER JOIN Ratings R ON R.MovieID = T.MovieID
INNER JOIN Tags_Master TM ON TM.ID = T.TagID
GROUP BY TM.Tag
HAVING COUNT(*) > 1000
ORDER BY 2 DESC, 3 DESC
LIMIT 25
```

| | tag<br>character varying (1000) | rating<br>numeric (10,2) | ratings<br>bigint |
|---|---|---|---|
| 1 | awesome! | 4.45 | 126 |
| 2 | Top 10 Movies | 4.45 | 126 |
| 3 | best-movie | 4.45 | 126 |
| 4 | excellent and worth while. | 4.45 | 126 |
| 5 | prison in some cases can make incarserated dependent and incomplete when term is serv... | 4.45 | 126 |
| 6 | stephan king | 4.45 | 126 |
| 7 | Frankly the most watchable film ever made | 4.45 | 63 |
| 8 | examines relationship between our lives and our freedom | 4.45 | 63 |
| 9 | meesa | 4.45 | 63 |
| 10 | patience | 4.45 | 63 |
| 11 | appeals to all audiences | 4.45 | 63 |
| 12 | Ali della libert+â-á | 4.45 | 63 |
| 13 | Phenomenal! | 4.45 | 63 |

Total rows: 25 of 25 Query complete 00:05:59.950

Fig. 17. Before optimization

- **Optimization Strategy:** Using the current composite primary key index on UserId and MovieID, we changed the join condition to include R.UserID = T.UserID in order to increase efficiency. The query execution time was slashed to a matter of seconds as a result.



```sql
SELECT TM.Tag, CAST(AVG(Rating) AS NUMERIC(10,2)) Rating, COUNT(*) Rat
FROM Tags T
INNER JOIN Ratings R ON R.MovieID = T.MovieID AND R.UserID = T.UserID
INNER JOIN Tags_Master TM ON TM.ID = T.TagID
GROUP BY TM.Tag
HAVING COUNT(*) > 1000
ORDER BY 2 DESC, 3 DESC
LIMIT 25
```

| | tag<br>character varying (1000) | rating<br>numeric (10,2) | ratings<br>bigint |
|---|---|---|---|
| 1 | thought-provoking | 4.38 | 1385 |
| 2 | classic | 4.34 | 1856 |
| 3 | psychology | 4.26 | 1696 |
| 4 | dark comedy | 4.24 | 1853 |
| 5 | cult film | 4.23 | 1043 |
| 6 | Surreal | 4.21 | 2456 |
| 7 | twist ending | 4.21 | 2320 |
| 8 | social commentary | 4.21 | 1398 |
| 9 | Atmospheric | 4.18 | 3020 |
| 10 | quirky | 4.14 | 1935 |
| 11 | Black comedy | 4.14 | 1183 |
| 12 | Dark | 4.13 | 1145 |
| 13 | satire | 4.13 | 1039 |
| 14 | Funny | 4.10 | 2189 |

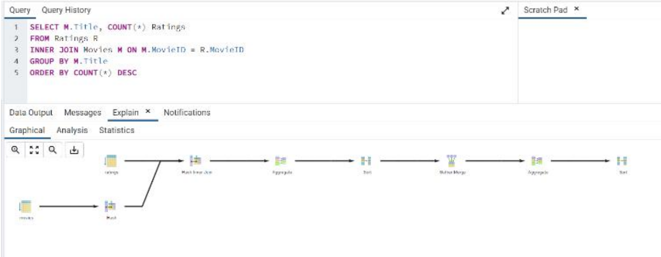
Total rows: 25 of 25 Query complete 00:00:01.599

Fig. 18. Post optimization

- **Alternative Approach:** Another potential solution could be to create a new table containing aggregated data for each movie. While this would significantly enhance query efficiency, it would involve a trade-off with the granularity of user data.

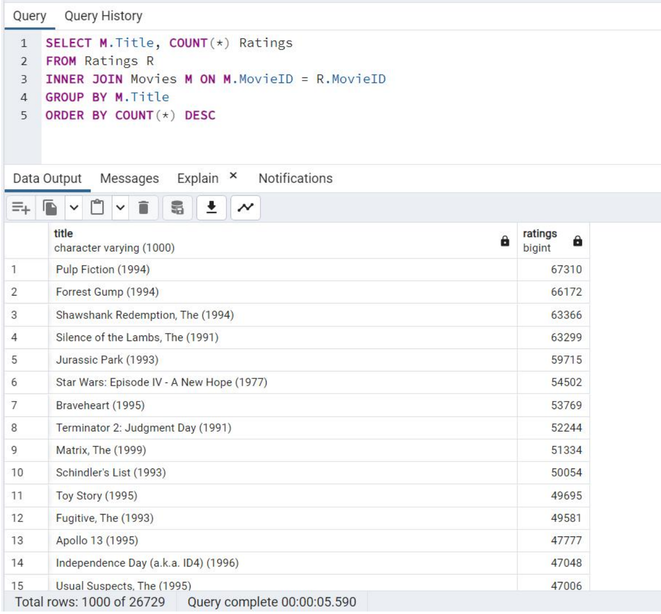
### 2. Movies with Most Number of Ratings

- **Initial Query and Issue:** The purpose of this query was to retrieve the highest-rated movies, and it was initially executing in roughly 5.5 seconds per execution. The main reason for the delay was found to be the lack of a specific index on the Ratings table's MovieID column.



```sql
SELECT M.Title, COUNT(*) Ratings
FROM Ratings R
INNER JOIN Movies M ON M.MovieID = R.MovieID
GROUP BY M.Title
ORDER BY COUNT(*) DESC
```

Fig. 19. Explain tool result on query



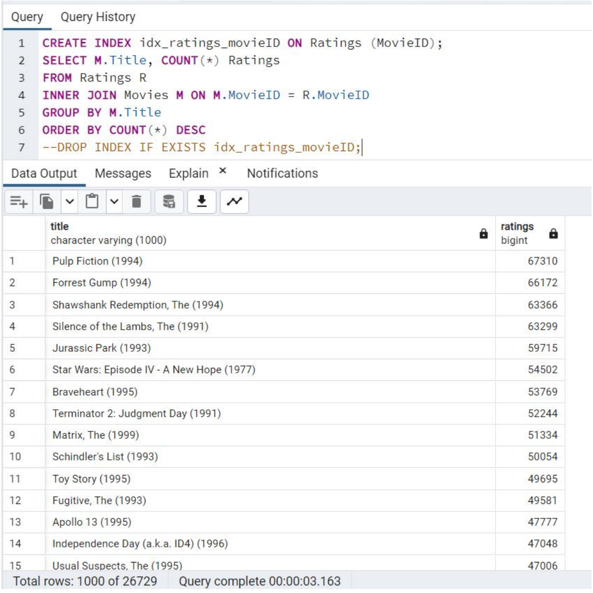
```sql
SELECT M.Title, COUNT(*) Ratings
FROM Ratings R
INNER JOIN Movies M ON M.MovieID = R.MovieID
GROUP BY M.Title
ORDER BY COUNT(*) DESC
```

| | title<br>character varying (1000) | ratings<br>bigint |
|---|---|---|
| 1 | Pulp Fiction (1994) | 67310 |
| 2 | Forrest Gump (1994) | 66172 |
| 3 | Shawshank Redemption, The (1994) | 63366 |
| 4 | Silence of the Lambs, The (1991) | 63299 |
| 5 | Jurassic Park (1993) | 59715 |
| 6 | Star Wars: Episode IV - A New Hope (1977) | 54502 |
| 7 | Braveheart (1995) | 53769 |
| 8 | Terminator 2: Judgment Day (1991) | 52244 |
| 9 | Matrix, The (1999) | 51334 |
| 10 | Schindler's List (1993) | 50054 |
| 11 | Toy Story (1995) | 49695 |
| 12 | Fugitive, The (1993) | 49581 |
| 13 | Apollo 13 (1995) | 47777 |
| 14 | Independence Day (a.k.a. ID4) (1996) | 47048 |
| 15 | Usual Suspects, The (1995) | 47006 |

Total rows: 1000 of 26729 Query complete 00:00:05.590

Fig. 20. Before Optimization

- **Optimization Strategy:** We created a non-clustered index on the MovieID column of the Ratings table. This optimization led to a 40% performance improvement, reducing the query execution time to approximately 3.5 seconds.



```sql
CREATE INDEX idx_ratings_movieID ON Ratings (MovieID);
SELECT M.Title, COUNT(*) Ratings
FROM Ratings R
INNER JOIN Movies M ON M.MovieID = R.MovieID
GROUP BY M.Title
ORDER BY COUNT(*) DESC
--DROP INDEX IF EXISTS idx_ratings_movieID;
```

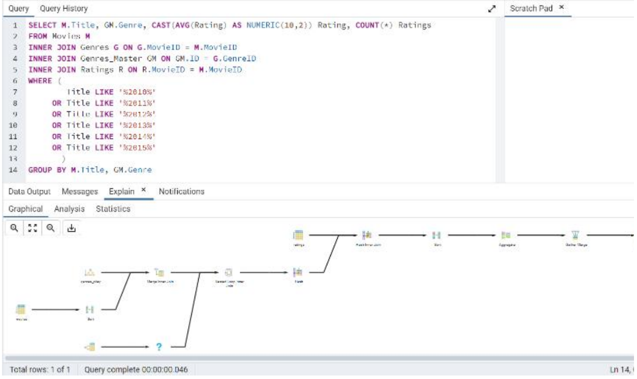
| | title<br>character varying (1000) | ratings<br>bigint |
|---|---|---|
| 1 | Pulp Fiction (1994) | 67310 |
| 2 | Forrest Gump (1994) | 66172 |
| 3 | Shawshank Redemption, The (1994) | 63366 |
| 4 | Silence of the Lambs, The (1991) | 63299 |
| 5 | Jurassic Park (1993) | 59715 |
| 6 | Star Wars: Episode IV - A New Hope (1977) | 54502 |
| 7 | Braveheart (1995) | 53769 |
| 8 | Terminator 2: Judgment Day (1991) | 52244 |
| 9 | Matrix, The (1999) | 51334 |
| 10 | Schindler's List (1993) | 50054 |
| 11 | Toy Story (1995) | 49695 |
| 12 | Fugitive, The (1993) | 49581 |
| 13 | Apollo 13 (1995) | 47777 |
| 14 | Independence Day (a.k.a. ID4) (1996) | 47048 |
| 15 | Usual Suspects, The (1995) | 47006 |

Total rows: 1000 of 26729   Query complete 00:00:03.163

Fig. 21. Post optimization

### 3. Fetching Movies Year-wise in Range

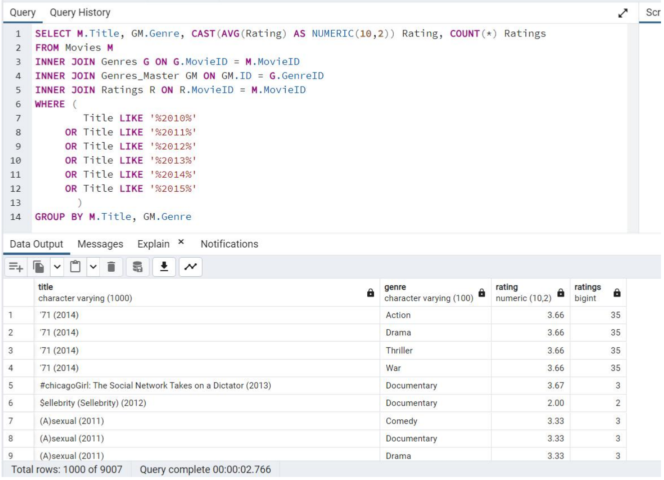
**Initial Query and Issue:** Using the LIKE operator on the movie title, the query sought to filter movies inside a specified range of years. This method had a substantial negative impact on performance because it required a full table scan.



```sql
SELECT M.Title, GM.Genre, CAST(AVG(Rating) AS NUMERIC(10,2)) Rating, COUNT(*) Ratings
FROM Movies M
INNER JOIN Genres G ON G.MovieID = M.MovieID
INNER JOIN Genres_Master GM ON GM.ID = G.GenreID
INNER JOIN Ratings R ON R.MovieID = M.MovieID
WHERE (
        Title LIKE '%2010%'
     OR Title LIKE '%2011%'
     OR Title LIKE '%2012%'
     OR Title LIKE '%2013%'
     OR Title LIKE '%2014%'
     OR Title LIKE '%2015%'
       )
GROUP BY M.Title, GM.Genre
```

Total rows: 1 of 1   Query complete 00:00:00.046   Ln 14,

Fig. 22. . Explain tool result on query



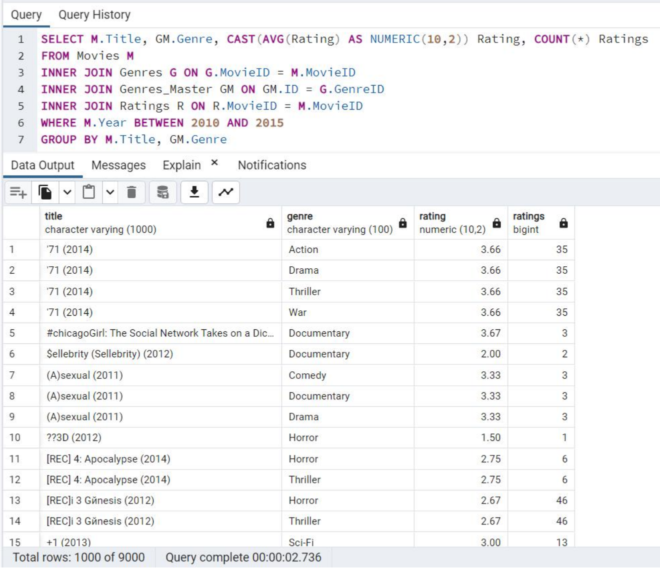
```sql
SELECT M.Title, GM.Genre, CAST(AVG(Rating) AS NUMERIC(10,2)) Rating, COUNT(*) Ratings
FROM Movies M
INNER JOIN Genres G ON G.MovieID = M.MovieID
INNER JOIN Genres_Master GM ON GM.ID = G.GenreID
INNER JOIN Ratings R ON R.MovieID = M.MovieID
WHERE (
        Title LIKE '%2010%'
     OR Title LIKE '%2011%'
     OR Title LIKE '%2012%'
     OR Title LIKE '%2013%'
     OR Title LIKE '%2014%'
     OR Title LIKE '%2015%'
       )
GROUP BY M.Title, GM.Genre
```

| | title<br>character varying (1000) | genre<br>character varying (100) | rating<br>numeric (10,2) | ratings<br>bigint |
|---|---|---|---|---|
| 1 | '71 (2014) | Action | 3.66 | 35 |
| 2 | '71 (2014) | Drama | 3.66 | 35 |
| 3 | '71 (2014) | Thriller | 3.66 | 35 |
| 4 | '71 (2014) | War | 3.66 | 35 |
| 5 | #chicagoGirl: The Social Network Takes on a Dictator (2013) | Documentary | 3.67 | 3 |
| 6 | $ellebrity (Sellebrity) (2012) | Documentary | 2.00 | 2 |
| 7 | (A)sexual (2011) | Comedy | 3.33 | 3 |
| 8 | (A)sexual (2011) | Documentary | 3.33 | 3 |
| 9 | (A)sexual (2011) | Drama | 3.33 | 3 |

Total rows: 1000 of 9007   Query complete 00:00:02.766

Fig. 23. Before Optimization

**Optimization Strategy:** We introduced a new Year column in the Movies table, separating the year from the title. This allowed for direct querying on the Year column, thereby avoiding full table scans and significantly improving the query's efficiency.



```sql
SELECT M.Title, GM.Genre, CAST(AVG(Rating) AS NUMERIC(10,2)) Rating, COUNT(*) Ratings
FROM Movies M
INNER JOIN Genres G ON G.MovieID = M.MovieID
INNER JOIN Genres_Master GM ON GM.ID = G.GenreID
INNER JOIN Ratings R ON R.MovieID = M.MovieID
WHERE M.Year BETWEEN 2010 AND 2015
GROUP BY M.Title, GM.Genre
```

| | title<br>character varying (1000) | genre<br>character varying (100) | rating<br>numeric (10,2) | ratings<br>bigint |
|---|---|---|---|---|
| 1 | '71 (2014) | Action | 3.66 | 35 |
| 2 | '71 (2014) | Drama | 3.66 | 35 |
| 3 | '71 (2014) | Thriller | 3.66 | 35 |
| 4 | '71 (2014) | War | 3.66 | 35 |
| 5 | #chicagoGirl: The Social Network Takes on a Dic... | Documentary | 3.67 | 3 |
| 6 | $ellebrity (Sellebrity) (2012) | Documentary | 2.00 | 2 |
| 7 | (A)sexual (2011) | Comedy | 3.33 | 3 |
| 8 | (A)sexual (2011) | Documentary | 3.33 | 3 |
| 9 | (A)sexual (2011) | Drama | 3.33 | 3 |
| 10 | ??3D (2012) | Horror | 1.50 | 1 |
| 11 | [REC] 4: Apocalypse (2014) | Horror | 2.75 | 6 |
| 12 | [REC] 4: Apocalypse (2014) | Thriller | 2.75 | 6 |
| 13 | [REC]i 3 Gйnesis (2012) | Horror | 2.67 | 46 |
| 14 | [REC]i 3 Gйnesis (2012) | Thriller | 2.67 | 46 |
| 15 | +1 (2013) | Sci-Fi | 3.00 | 13 |

Total rows: 1000 of 9000   Query complete 00:00:02.736

Fig. 24. Post Optimization

These optimizations highlight the importance of understanding the underlying database structure and query execution plan. By analyzing and modifying our queries and database schema, we achieved significant performance improvements, ensuring faster and more efficient data retrieval. This exercise underscores the critical role of database tuning in managing large datasets and provides valuable insights into practical database optimization techniques.

## IV. Database Deployment (Bonus Task)

The task involved creating an interactive web application using Streamlit, a popular Python library for building web applications, to visualize and display query results from a PostgreSQL database. The primary focus was on a movie database, aptly named "Movie Mania", which allows users to search for movies based on different criteria such as title, year, rating, and genre.

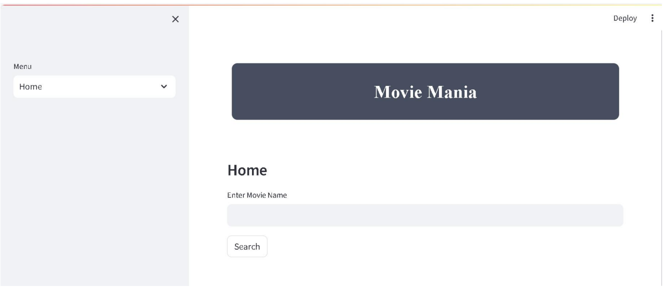
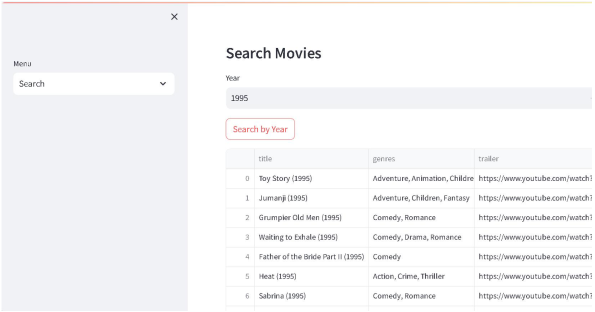


Fig. 24. Home Page

**Technical Implementation**

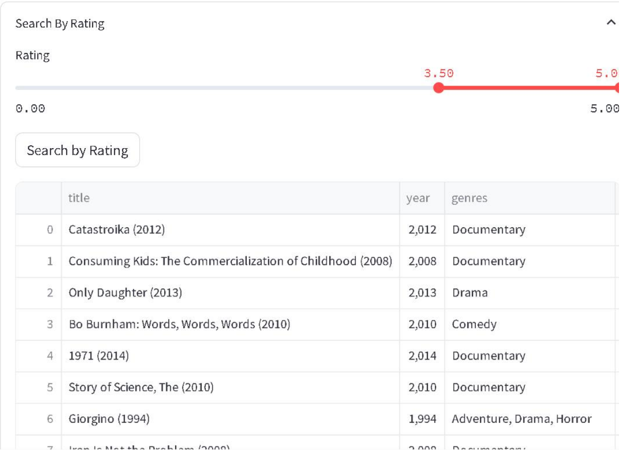
1. **Streamlit Integration**: Streamlit was chosen for its simplicity and effectiveness in creating data-driven web applications. The application's layout includes a sidebar for navigation and different sections for user interaction.
2. **Database Connection**: The application connects to a PostgreSQL database using SQLAlchemy, a Python SQL toolkit, ensuring a secure and efficient link for querying the database.

3. **User Interface Design**: The web application features an engaging user interface, with a banner and various input options like text input, number input, and sliders. This design enhances user experience, making it intuitive to search for movies.
4. **Dynamic Querying**: The application dynamically constructs SQL queries based on user input. These queries fetch movie data like title, genres, trailer links, average ratings, and number of ratings. This approach allows for real-time, customized data retrieval.



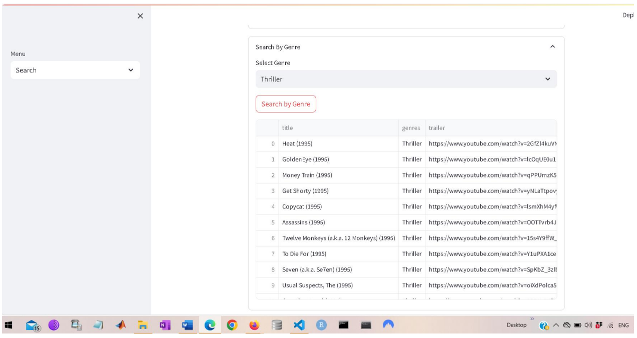
| | title | genres | trailer |
|---|---|---|---|
| 0 | Toy Story (1995) | Adventure, Animation, Childre | https://www.youtube.com/watch? |
| 1 | Jumanji (1995) | Adventure, Children, Fantasy | https://www.youtube.com/watch? |
| 2 | Grumpier Old Men (1995) | Comedy, Romance | https://www.youtube.com/watch? |
| 3 | Waiting to Exhale (1995) | Comedy, Drama, Romance | https://www.youtube.com/watch? |
| 4 | Father of the Bride Part II (1995) | Comedy | https://www.youtube.com/watch? |
| 5 | Heat (1995) | Action, Crime, Thriller | https://www.youtube.com/watch? |
| 6 | Sabrina (1995) | Comedy, Romance | https://www.youtube.com/watch? |

Fig. 25.Search by year



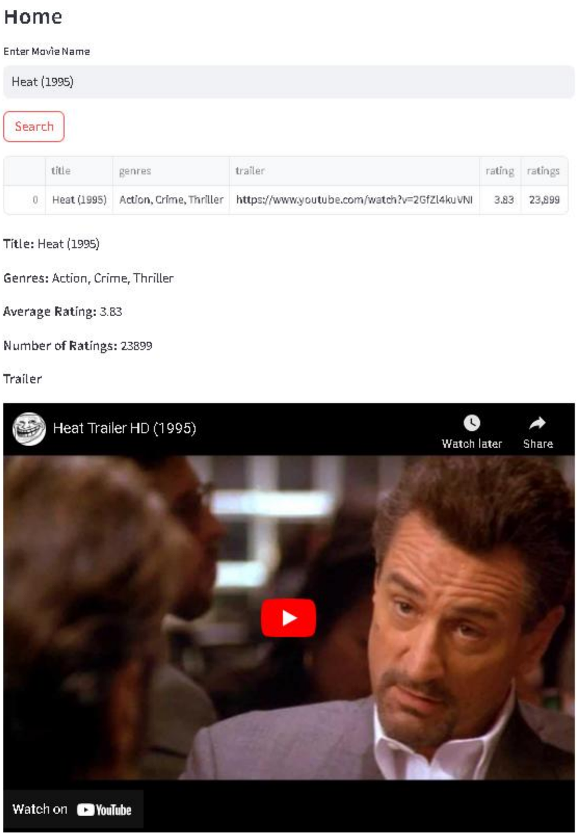
| | title | year | genres |
|---|---|---|---|
| 0 | Catastroika (2012) | 2,012 | Documentary |
| 1 | Consuming Kids: The Commercialization of Childhood (2008) | 2,008 | Documentary |
| 2 | Only Daughter (2013) | 2,013 | Drama |
| 3 | Bo Burnham: Words, Words, Words (2010) | 2,010 | Comedy |
| 4 | 1971 (2014) | 2,014 | Documentary |
| 5 | Story of Science, The (2010) | 2,010 | Documentary |
| 6 | Giorgino (1994) | 1,994 | Adventure, Drama, Horror |

Fig. 26.Search by rating



| | title | genres | trailer |
|---|---|---|---|
| 0 | Heat (1995) | Thriller | https://www.youtube.com/watch?v=2GfZl4kuVN |
| 1 | GoldenEye (1995) | Thriller | https://www.youtube.com/watch?v=lcOqUE0u1 |
| 2 | Money Train (1995) | Thriller | https://www.youtube.com/watch?v=qPPUmzK5 |
| 3 | Get Shorty (1995) | Thriller | https://www.youtube.com/watch?v=yNLaTtpov |
| 4 | Copycat (1995) | Thriller | https://www.youtube.com/watch?v=lsmXhM4yf |
| 5 | Assassins (1995) | Thriller | https://www.youtube.com/watch?v=OOTTvrb4J |
| 6 | Twelve Monkeys (a.k.a. 12 Monkeys) (1995) | Thriller | https://www.youtube.com/watch?v=15s4Y9ffW_ |
| 7 | To Die For (1995) | Thriller | https://www.youtube.com/watch?v=Y1uPXA1ce |
| 8 | Seven (a.k.a. Se7en) (1995) | Thriller | https://www.youtube.com/watch?v=SpKbZ_3zl |
| 9 | Usual Suspects, The (1995) | Thriller | https://www.youtube.com/watch?v=oiXdPolca5 |

Fig. 27.Search by genre



| | title | genres | trailer | rating | ratings |
|---|---|---|---|---|---|
| 0 | Heat (1995) | Action, Crime, Thriller | https://www.youtube.com/watch?v=2GfZl4kuVNI | 3.83 | 23,899 |

**Title:** Heat (1995)

**Genres:** Action, Crime, Thriller

**Average Rating:** 3.83

**Number of Ratings:** 23899

Trailer

Fig. 28.Search by title

5. **Data Visualization**: Results from queries are displayed in a tabular format, providing a clear and organized presentation of data. Additional features like movie trailers are embedded directly into the application for an enriched user experience.
6. **Responsive Search Features**: The application offers various search functionalities:
   - **Search by Title**: Allows users to enter a movie name and retrieve relevant data.
   - **Search by Year**: Users can select a year to find movies released in that year.
   - **Search by Rating**: A slider to choose a rating range, returning movies within that rating spectrum.
   - **Search by Genre**: Users can select from a list of genres to find movies in the selected category.
7. **Error Handling and User Feedback**: The application provides user feedback and warnings, such as when no movies are found or a movie name is not entered.
8. **Accessibility and Performance**: Streamlit's lightweight framework ensures the application is accessible and performs efficiently, even with large datasets.

## REFERENCES

[1] https://grouplens.org/datasets/movielens/20m/

[2] https://grouplens.org/datasets/movielens/20m-youtube/