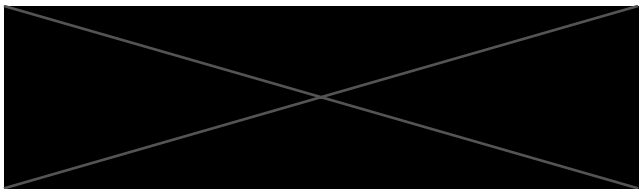
[redacted]

Movie Recommendation Database Design



I. PROBLEM STATEMENT

The Movie Recommendation Database seeks to create a user-centric movie recommendation system using Movie Lens dataset. This system will offer personalized movie suggestions based on user preferences, demographics, and past behavior while ensuring a diverse range of content. It will include an intuitive and scalable user interface and integrate user ratings and reviews. Privacy and security measures will safeguard user data, and feedback mechanisms will enable continuous improvement. Our project wants to make things better by creating a strong database system. This won't just help with entering and finding data but will also open possibilities for advanced analysis and insights. By fixing the issues with Excel, we hope to make sure our movie data is accurate, promote teamwork, and make managing movie information much smoother. Our project, the Movie Recommendation Database, aims to replace Excel with a dedicated database to build an advanced movie recommendation system. Excel's limitations in handling large datasets hinder the creation of a user-centric recommendation system that considers preferences, demographics, and past behavior. The transition to a database is crucial for data accuracy, advanced analytics, and improved user experience.

A. Background

With the growing number of movies available, there's a need for better recommendation systems. Excel struggles with big datasets, hindering accurate suggestions. Our database tackles this issue.

B. Contribution

Our project sets a new standard for user-friendly movie recommendation. Moving from Excel to a dedicated database not only fixes existing problems but also allows for advanced analysis, insights, and efficient movie information management, enhancing the overall user experience.

II. TARGET USER

A. Users of the Database

Anyone who loves movies, does research, or builds apps related to movies can benefit from our database. It's like a treasure trove of movie information for people who want to know trends, study what viewers like, or create cool apps connected to the movie world.

B. Database Administration

Taking care of the database will be done by a team of experts in managing data and databases. They'll make sure the information is safe, update it when needed, and make sure everything runs smoothly. In a real-life situation, a movie database manager might work with a group of analysts, developers, and content experts to keep the database in good shape.

III. REAL LIFE SCENARIO

Consider a real-life scenario involving the Movie Recommendation Database:
Imagine Bob, a 23-year-old who enjoys watching movies in his free time. He subscribes to a popular streaming service and often finds himself overwhelmed by the vast catalog of movies available. He is interested in a variety of genres, from action and thriller to romantic comedies. Using the Movie Recommendation Database, Bob creates a profile, rates some of his favorite films, and provides information about his preferences. The recommendation system then analyzes his data and suggests a list of movies tailored to his taste, offering both popular blockbusters and lesser-known gems in his preferred genres. Bob uses the system's user-friendly interface to explore these recommendations, read reviews from other users, and ultimately decides on a movie to watch with confidence, knowing that it aligns with his cinematic preferences. The system continues to learn and adapt, ensuring that Sujith’s

future movie choices remain as enjoyable as her first experience.

IV. DATABASE OVERVIEW

A. *Overview*

The aim of this project is to design and implement a movie recommendation system using the MovieLens dataset. This system will help users discover movies based on their preferences, viewing history, and demographic information. The MovieLens dataset provides a rich source of movie ratings, user data, and movie details that can be leveraged for building a personalized recommendation engine.

B. *Dataset*

The MovieLens dataset includes information on users, movies, ratings, genres, and more. This real-world dataset will be used to create a robust and diverse movie recommendation system. In case of constraints accessing the real dataset, a program-generated dataset can be used to simulate user preferences and interactions.

C. *Queries:*

1) User-based Recommendations: Recommend movies based on a user's historical ratings and preferences. Example Query: "What are the top movies recommended for User_123?"

2) Genre-based Recommendations: Recommend movies based on a user's preferred genre. Example Query: "List the top-rated action movies for User_456."

3) Popular Movies: Identify the most popular movies based on overall ratings. Example Query: "What are the top-rated movies in the database?"

4) New Releases: Show recently released movies. Example Query: "List movies released in the last month."

5) User Interactions: View a user's ratings and interactions. Example Query: "Retrieve all ratings given by User_789."

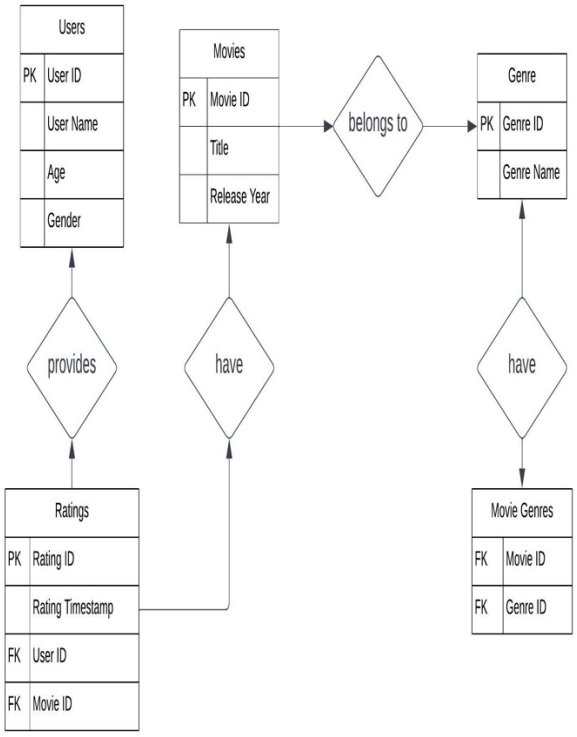
6) Updates: Users can add new ratings for movies they've recently watched. Users can update their profile information (e.g., age, gender). New movies can be added to the database.

D. *Application:*

The application will provide a user-friendly interface where users can log in, rate movies, update their profiles, and receive personalized movie recommendations. The recommendation engine will continuously evolve based on user interactions, ensuring that recommendations remain relevant over time.

This use case provides a comprehensive exploration of SQL operations, including complex queries for recommendation generation and database updates to reflect user interactions and new movie releases.

V. ER DIAGRAM



A. Database Design, Relations & Sample Records in Tables:

There are 5 tables in total.

Users table consists of following attributes

Attribute Name	Data Type	Description	Constraint
UserID	INT	Unique ID for User	Primary Key
UserName	VARCHAR(256)	Name of the user	Not Null
Age	INT	Age of user	
Gender	VARCHAR(10)	Gender of User	

	UserID [PK] integer	UserName character varying (256)	Age integer	Gender character varying (10)
1	120158	User_120158	44	Male
2	48080	User_48080	34	Male
3	125691	User_125691	31	Male
4	112131	User_112131	43	Male
5	126976	User_126976	29	Male
6	72324	User_72324	42	Female
7	93384	User_93384	18	Male
8	55249	User_55249	24	Female
9	85226	User_85226	33	Female
10	100264	User_100264	43	Male
11	92091	User_92091	23	Male
12	54313	User_54313	42	Female
13	112018	User_112018	53	Male

Movies table consists of following attributes.

Attribute Name	Data Type	Description	Constraint
MovieID	INT	Unique ID for Movie	Primary Key
Title	VARCHAR(256)	Title of the movie	Not Null
ReleaseYear	INT	Year in which movie released	Not Null
Genre	VARCHAR(256)	Genre of Movie	Not Null

	MovieID [PK] integer	Title character varying (255)	Release Year integer	Genre character varying (255)
1	1	Toy Story	1995	Comedy
2	2	Jumanji	1995	Comedy
3	3	Grumpier Old Men	1995	Comedy
4	4	Waiting to Exhale	1995	Comedy
5	5	Father of the Bride Part II	1995	Drama
6	6	Heat	1995	Sci-Fi
7	7	Sabrina	1995	Action
8	8	Tom and Huck	1995	Thriller
9	9	Sudden Death	1995	Comedy
10	10	GoldenEye	1995	Sci-Fi
11	11	American President, The	1995	Thriller
12	12	Dracula: Dead and Loving It	1995	Comedy
13	13	Balto	1995	Sci-Fi
14	14	Nixon	1995	Action
15	15	Cutthroat Island	1995	Comedy
16	16	Casino	1995	Action
17	17	Sense and Sensibility	1995	Comedy
18	18	Four Rooms	1995	Sci-Fi
19	19	Ace Ventura: When Nature Calls	1995	Comedy
20	20	Money Train	1995	Thriller
21	21	Get Shorty	1995	Sci-Fi
22	22	Copycat	1995	Thriller
23	23	Assassins	1995	Drama

Genres table consists of following attributes

Attribute Name	Data Type	Description	Constraint
GenreID	INT	Unique ID for Genre	Primary Key
GenreName	VARCHAR	Genre of Movie	Not Null

	GenreID [PK] integer	GenreName character varying
1	1	Action
2	2	Drama
3	3	Comedy
4	4	Sci-Fi
5	5	Thriller
6	6	Romance
7	7	Horror
8	8	Fantasy
9	9	Adventure

Ratings table consists of following attributes.

Attribute Name	Data Type	Description	Constraint
RatingID	INT	Unique ID for Rating	Primary Key
UserID	INT	Unique ID for User	Foreign Key, On Delete Cascade, On Update Cascade
MovieID	INT	Unique ID for Movie	Foreign Key, On Delete Cascade, On Update Cascade
Rating	DECIMAL(2,1)	Rating given to a movie	Not Null
Timestamp	Timestamp		

	RatingId [PK] integer	UserID integer	MovieID integer	Ratings numeric (2,1)	TimeStamp timestamp without time zone
1	1	8931	128173	1.6	2023-10-02 16:08:49.308527
2	2	58584	43549	4.4	2022-12-27 16:08:49.308604
3	3	29885	37	4.4	2022-11-27 16:08:49.308625
4	4	29634	26937	2.7	2023-02-24 16:08:49.308642
5	5	103208	82003	3.2	2023-07-30 16:08:49.30866
6	6	33680	7133	2.6	2023-05-26 16:08:49.308679
7	7	116849	5427	2.3	2023-07-21 16:08:49.308697
8	8	78098	96160	1.5	2023-06-24 16:08:49.308714
9	9	13595	85365	2.4	2023-10-14 16:08:49.308744
10	10	86131	50585	3.6	2023-09-30 16:08:49.308761

Movie_Genres table consists of following attributes

Attribute Name	Data Type	Description	Constraint
MovieID	INT	Unique ID for Movie	Foreign Key, Primary Key, On Delete Cascade, On Update Cascade
GenreID	INT	Unique ID for Genre	Foreign Key, Primary Key, On Delete Cascade, On Update Cascade

	MovieID [PK] integer	GenreID [PK] integer
1	1	7
2	1	1
3	1	9
4	1	2
5	1	8
6	1	6
7	2	5
8	2	3
9	2	4
10	3	3

B. Constraints

1) NOT NULL Constraint:

a) Users Table:
- UserName
  - *Significance:* Ensures that a name is provided for each user.
  - *NOT NULL Constraint:* Guarantees that the UserName attribute cannot be empty.

b) Movies Table:
- Title:
  - *Significance:* Ensures that a title is provided for each movie.
  - *NOT NULL Constraint:* Guarantees that the Title attribute cannot be empty.
- ReleaseYear:
  - *Significance:* Ensures that a release year is provided for each movie.
  - *NOT NULL Constraint:* Guarantees that the ReleaseYear attribute cannot be empty.
- Genre:
  - *Significance:* Ensures that a genre is provided for each movie.
  - *NOT NULL Constraint:* Guarantees that the Genre attribute cannot be empty.

c) Genres Table:
- GenreName:
  - *Significance:* Ensures that a name is provided for each genre.
  - *NOT NULL Constraint:* Guarantees that the GenreName attribute cannot be empty.

d) Ratings Table:
- Rating:
  - *Significance:* Ensures that a rating is provided for each record.
  - *NOT NULL Constraint:* Guarantees that the Rating attribute cannot be empty.

The **NOT NULL** constraint ensures that essential information is provided for each record, preventing the introduction of incomplete or invalid data into the database. It contributes to data integrity by enforcing the presence of values in critical attributes.

2) Referential Integrity Constraint, On Delete Cascade, On Update Cascade Constraint:

a) Ratings Table:
- UserID (Foreign Key):
  - *Referential Integrity:* This foreign key establishes a relationship between the Ratings table and the Users table based

on the UserID attribute.
    - *ON DELETE CASCADE:* If a user record in the Users table is deleted, all corresponding ratings in the Ratings table for that user will also be deleted, maintaining consistency.
    - *ON UPDATE CASCADE:* If the UserID of a user in the Users table is updated, the corresponding UserID in the Ratings table will be updated accordingly.
- MovieID (Foreign Key):
    - *Referential Integrity:* This foreign key establishes a relationship between the Ratings table and the Movies table based on the MovieID attribute.
    - *ON DELETE CASCADE:* If a movie record in the Movies table is deleted, all corresponding ratings in the Ratings table for that movie will also be deleted, maintaining consistency.
    - *ON UPDATE CASCADE:* If the MovieID of a movie in the Movies table is updated, the corresponding MovieID in the Ratings table will be updated accordingly.

b) Movie_Genres Table:
- MovieID (Foreign Key):
    - *Referential Integrity:* This foreign key establishes a relationship between the Movie_Genres table and the Movies table based on the MovieID attribute.
    - *ON DELETE CASCADE:* If a movie record in the Movies table is deleted, all corresponding records in the Movie_Genres table for that movie will also be deleted, maintaining consistency.
    - *ON UPDATE CASCADE:* If the MovieID of a movie in the Movies table is updated, the corresponding MovieID in the Movie_Genres table will be updated accordingly.
- GenreID (Foreign Key):
    - *Referential Integrity:* This foreign key establishes a relationship between the Movie_Genres table and the Genres table based on the GenreID attribute.
    - *ON DELETE CASCADE:* If a genre record in the Genres table is deleted, all corresponding records in the Movie_Genres table for that genre will also be deleted, maintaining consistency.
    - *ON UPDATE CASCADE:* If the GenreID of a genre in the Genres table is updated, the corresponding GenreID in the Movie_Genres table will be updated accordingly.

These constraints and cascading actions are essential for maintaining referential integrity in the database. They ensure that

relationships between tables remain valid, and any changes to primary key values are properly reflected in related foreign key columns, preventing inconsistencies in the data.

3) PRIMARY KEY Constraint:

a) Users Table:
- UserID (Primary Key):
  - *Significance:* Uniquely identifies each user.
  - *PRIMARY KEY Constraint:* Ensures that each UserID is unique and serves as the primary means of identifying users in the Users table.

b) Movies Table:
- MovieID (Primary Key):
  - *Significance:* Uniquely identifies each movie.
  - *PRIMARY KEY Constraint:* Ensures that each MovieID is unique and serves as the primary means of identifying movies in the Movies table.

c) Genres Table:
- GenreID (Primary Key):
  - *Significance:* Uniquely identifies each genre.
  - *PRIMARY KEY Constraint:* Ensures that each GenreID is unique and serves as the primary means of identifying genres in the Genres table.

d) Ratings Table:
- RatingID (Primary Key):
  - *Significance:* Uniquely identifies each rating.
  - *PRIMARY KEY Constraint:* Ensures that each RatingID is unique and serves as the primary means of identifying ratings in the Ratings table.

e) Movie_Genres Table:
- MovieID (Primary Key):
  - *Significance:* Uniquely identifies each movie within the Movie_Genres table.
  - *PRIMARY KEY Constraint:* Ensures that each MovieID is unique within the context of the Movie_Genres table.
- GenreID (Primary Key):
  - *Significance:* Uniquely identifies each genre within the Movie_Genres table.
  - *PRIMARY KEY Constraint:* Ensures that each GenreID is unique within the context of the Movie_Genres table.

The **PRIMARY KEY** constraint plays a crucial role in ensuring data integrity by preventing the insertion of duplicate or null values in key columns. It uniquely identifies each record within a table, facilitating efficient data retrieval and maintaining the integrity of relational connections between tables.

VI. DATA LOADING / CSV FILES GENERATING

We have executed python scripts to generate structured data from Movie Lens Dataset to insert into the respective tables

```python
import pandas as pd
import random

# Load the MovieLens 20M dataset or a sample of it (adjust the path accordingly)
# movielens_path = 'path/to/movielens-20m-dataset'
ratings_df = pd.read_csv('movielens/rating.csv')

# Extract unique user IDs
unique_user_ids = ratings_df['userId'].unique()

# Sample 100 user IDs
sample_user_ids = random.sample(list(unique_user_ids), 2000)

# Create a DataFrame with sample user data
sample_users_df = pd.DataFrame({
    'UserID': sample_user_ids,
    'UserName': [f'User_{i}' for i in sample_user_ids],
    'Age': [random.randint(18, 60) for _ in range(2000)],
    'Gender': ['Male' if random.choice([True, False]) else 'Female' for _ in range(2
})

# Save the DataFrame to a CSV file
sample_users_df.to_csv('sample_users.csv', index=False)

print("CSV file 'sample_users.csv' generated successfully.")

```

```
CSV file 'sample_users.csv' generated successfully.
```

```python
import pandas as pd
import random
import re  # Import the regular expression module

# Load the MovieLens 20M dataset or a sample of it (adjust the path accordingly)
movies_df = pd.read_csv('movielens/movie.csv')  # Assuming there's a 'movies.csv' file in

# Sample data for Movies table
sample_movies_data = []

# Regular expression pattern to extract the title and year
pattern = re.compile(r'^(?P<Title>.*?)\s\((?P<Year>\d{4})\)$')

for _, row in movies_df.iterrows():
    # Use regex to extract title and year from the original title
    match = pattern.match(row['title'])

    # If the pattern matches, extract title and year
    if match:
        title = match.group('Title')
        year = int(match.group('Year'))
    else:
        # If the pattern doesn't match, use the original title and a random year
        title = row['title']
        year = random.randint(1990, 2023)  # Adjust the range based on your data

    sample_movies_data.append({
        'MovieID': row['movieId'],
        'Title': title,
        'ReleaseYear': year,
        'Genre': random.choice(['Action', 'Drama', 'Comedy', 'Sci-Fi', 'Thriller'])
    })

# Create a DataFrame with sample movies data
sample_movies_df = pd.DataFrame(sample_movies_data)

# Save the DataFrame to a CSV file
sample_movies_df.to_csv('sample_movies.csv', index=False)

print("CSV file 'sample_movies.csv' generated successfully.")

```

```
CSV file 'sample_movies.csv' generated successfully.
```

```python
import pandas as pd
import random
from datetime import datetime, timedelta

# Load sample data for Movies and Users tables
movies_df = pd.read_csv('sample_movies.csv')
users_df = pd.read_csv('sample_users.csv')

# Sample data for Ratings table
sample_ratings_data = []

# Counter for RatingID
rating_id_counter = 1

for _ in range(5000):  # Adjust the number of rows as needed
    sample_ratings_data.append({
        'RatingID': rating_id_counter,
        'UserID': random.choice(users_df['UserID']),
        'MovieID': random.choice(movies_df['MovieID']),
        'Rating': round(random.uniform(1, 5), 2),
        'Timestamp': datetime.now() - timedelta(days=random.randint(1, 365))
    })

    rating_id_counter += 1

# Create a DataFrame with sample ratings data
sample_ratings_df = pd.DataFrame(sample_ratings_data)

# Save the DataFrame to a CSV file
sample_ratings_df.to_csv('sample_ratings.csv', index=False)

print("CSV file 'sample_ratings.csv' generated successfully.")

```

```
CSV file 'sample_ratings.csv' generated successfully.
```

```python
import pandas as pd

# Sample data for Genres table
sample_genres_data = []

# Assuming you have predefined genre names or you can modify this list
genre_names = ['Action', 'Drama', 'Comedy', 'Sci-Fi', 'Thriller', 'Romance', 'Horror', 'Fantasy', 'Adver

for genre_id, genre_name in enumerate(genre_names, start=1):
    sample_genres_data.append({
        'GenreID': genre_id,
        'GenreName': genre_name
    })

# Create a DataFrame with sample genres data
sample_genres_df = pd.DataFrame(sample_genres_data)

# Save the DataFrame to a CSV file
sample_genres_df.to_csv('sample_genres.csv', index=False)

print("CSV file 'sample_genres.csv' generated successfully.")

```

```
CSV file 'sample_genres.csv' generated successfully.
```

```python
import pandas as pd

# Load sample data for Movies and Genres tables
movies_df = pd.read_csv('sample_movies.csv')
genres_df = pd.read_csv('sample_genres.csv')

# Sample data for Movies_Genres table
sample_movie_genres_data = []

for _, movie_row in movies_df.iterrows():
    # Randomly assign genres to movies
    movie_genres = random.sample(list(genres_df['GenreID']), k=random.randint(1, len(genre

    for genre_id in movie_genres:
        sample_movie_genres_data.append({
            'MovieID': movie_row['MovieID'],
            'GenreID': genre_id
        })

# Create a DataFrame with sample Movie_Genres data
sample_movie_genres_df = pd.DataFrame(sample_movie_genres_data)

# Save the DataFrame to a CSV file
sample_movie_genres_df.to_csv('sample_movie_genres.csv', index=False)

print("CSV file 'sample_movie_genres.csv' generated successfully.")

```

```
CSV file 'sample_movie_genres.csv' generated successfully.
```

VII. BCNF

Users:

- UserID -> UserName, Age, Gender
- UserID is the primary key, and there are no partial dependencies.

Movies:

- MovieID -> Title, ReleaseYear, Genre
- MovieID is the primary key, and there are no partial dependencies.

Ratings:

- RatingID -> UserID, MovieID, Rating, Timestamp
- RatingID is the primary key, and there are no partial dependencies.

Genres:

- GenreID -> GenreName
- GenreID is the primary key, and there are no partial dependencies.

Movies_Genres:

- (MovieID, GenreID) -> (Attributes of MovieID, Attributes of GenreID)
- The combination of (MovieID, GenreID) is the primary key, and there are no partial dependencies.

A. Justification about BCNF

For a relation to be in Boyce-Codd Normal Form (BCNF), it must meet the following criteria:

1. Every non-prime attribute is fully functionally dependent on the primary key.
2. There are no non-trivial functional dependencies of a proper subset of the primary key.

Looking at the functional dependencies identified above, all relations satisfy these criteria:

- In Users, Movies, Ratings, and Genres, all non-prime attributes are fully functionally dependent on their respective primary keys.
- In Movies_Genres, the combination of (MovieID, GenreID) serves as the primary key, and all attributes are fully functionally dependent on this combination.

Therefore, based on the identified functional dependencies, it can be concluded that the relations are already in Boyce-Codd Normal Form (BCNF).

VIII. CHALLENGES / INDEXING

A. Handling Larger Dataset Challenges

1) Slow Query Performance :
   a. One challenge faced with the larger dataset was the potential for slower query performance, especially when executing complex queries involving joins and aggregations.
   b. The sheer volume of data in tables like "Ratings" and "Movies" could lead to longer query execution times.

2) Optimization of Aggregation Queries :
   a. Aggregation queries, such as those calculating rating variability or user engagement, could be resource-intensive with a large dataset.
   b. There was a need to optimize these queries to ensure they completed within a reasonable time frame.

B. Adoption of Indexing Concepts:

1) Indexing on Join Columns:
   a. To address the slow query performance, indexing concepts were adopted, particularly on columns used in join conditions.
   b. Indexes were created on columns like "Movies"."MovieID" and "Ratings"."MovieID" to enhance the efficiency of join operations.

2) Indexing for Aggregation Conditions:

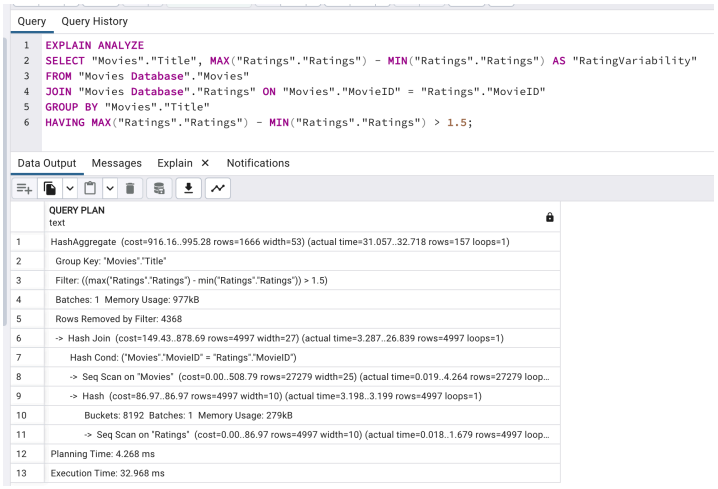
   Indexes were considered for columns involved in conditions within aggregation queries. For instance, indexing on "Ratings"."Ratings" could benefit queries calculating rating variability.

C. Results

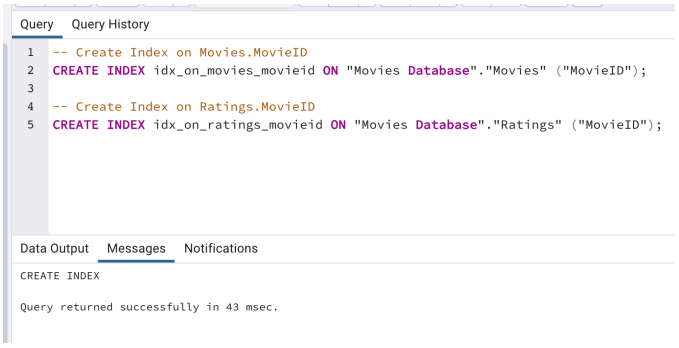
The adoption of indexing concepts led to notable improvements in query performance, particularly for join and aggregation operations. Regular monitoring of query execution plans and fine-tuning the indexing strategy played a key role in achieving optimized database performance with the larger dataset.

D. Explanation With Example

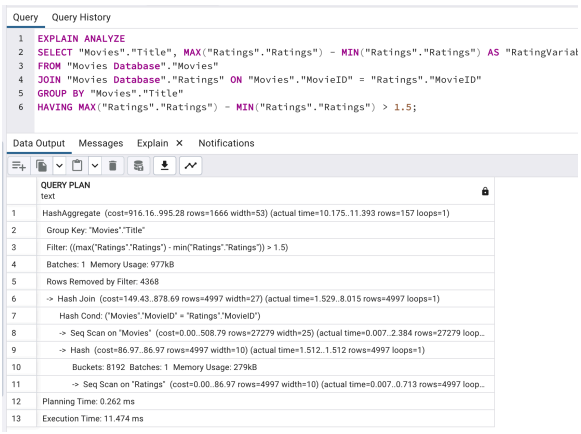
1) Before Indexing:
   a. The first **EXPLAIN** statement shows the query execution plan before creating indexes.
   b. This helps illustrate the potential performance issues associated with a large dataset and lack of indexes on relevant columns and we could see the execution time is 32 ms.



2) Create Indexes:
   a. Two indexes are created on columns used in join conditions: "Movies"."MovieID" and "Ratings"."MovieID."



3) After Indexing:
   a. The second **EXPLAIN** statement shows the query execution plan after creating indexes.
   b. The goal is to observe any improvements in query performance, especially in terms of index usage and now we could see reduction in execution time substantially.

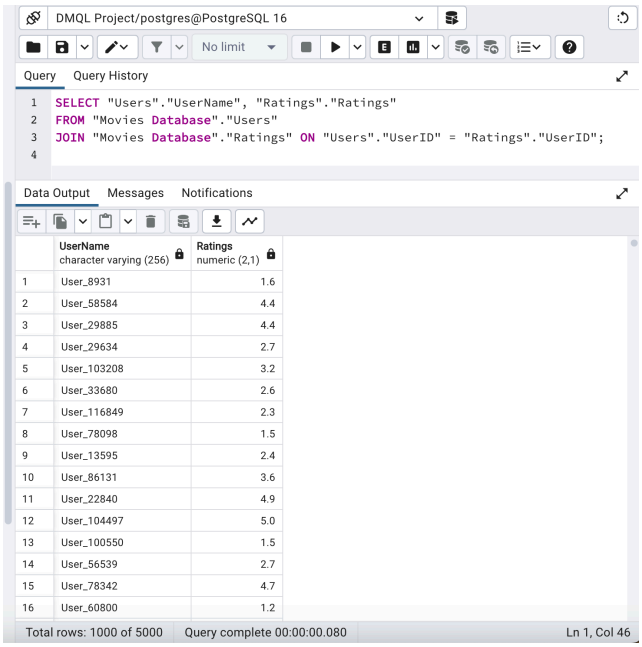


IX. QUERIES

A) SELECT Query 1 (Using Join)

```
SELECT "Users"."UserName",
"Ratings"."Rating"
FROM "Movies Database"."Users"
JOIN "Movies Database"."Ratings" ON
"Users"."UserID" = "Ratings"."UserID";
```

1. Query Analysis : This query performs a join operation between the "Users" and "Ratings" tables based on the "UserID" column. The resulting dataset includes the "UserName" from the "Users" table and the corresponding "Rating" from the "Ratings" table. **This query simple retrieves the usernames and corresponding ratings from the Users and Ratings tables**, providing a list of users and their associated movie ratings.



2. *Execution Steps:*

   - The database engine sequentially scans the "Users" table.
   - It then performs a join operation with the "Ratings" table based on matching "UserID" values.
   - Rows where the "UserID" values match are selected.
   - The final result includes the "UserName" from the "Users" table and the associated "Rating" from the "Ratings" table.

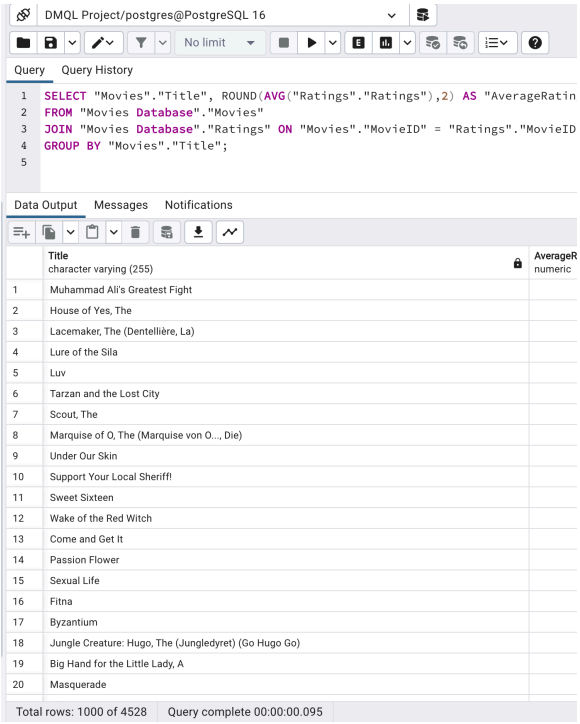
3. *Query Execution Time:* This is a relatively straightforward query with a moderate execution time. The cost of the query is influenced by the sequential scan on the "Users" table and the subsequent join operation. Overall, the query is expected to execute efficiently, and the total cost is reasonably low.

B) SELECT Query 2 (Using JOIN & GROUPBY)

```
SELECT "Movies"."Title",
AVG("Ratings"."Rating") AS "AverageRating"
FROM "Movies Database"."Movies"
JOIN "Movies Database"."Ratings" ON
"Movies"."MovieID" = "Ratings"."MovieID"
GROUP BY "Movies"."Title";
```

1) Query Analysis:

This query calculates the average rating for each movie in the "Movies Database." It performs a join operation between the "Movies" and "Ratings" tables based on matching "MovieID" values. The result is grouped by the movie title, and the average rating is computed for each group.



2) Execution Steps:

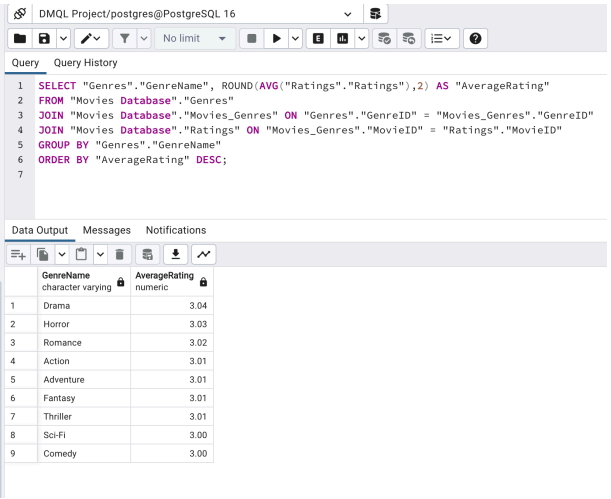
- Sequential scan of the "Movies" table.
- Join operation with the "Ratings" table based on matching "MovieID" values.
- Selection of rows where "MovieID" values match.
- Grouping of the result set by the "Title" column.
- Computation of the average rating for each group, labeled as "AverageRating."

3) Query Execution Time: Moderate execution time due to the join and group-by operations. The use of aggregate functions may impact performance, especially with larger datasets.

C) SELECT Query 3 (Using JOIN, GROUP BY & ORDER BY):

```
SELECT "Genres"."GenreName",
ROUND(AVG("Ratings"."Rating"),2) AS
"AverageRating"
FROM "Movies Database"."Genres"
JOIN "Movies Database"."Movies_Genres" ON
"Genres"."GenreID" =
"Movies_Genres"."GenreID"
JOIN "Movies Database"."Ratings" ON
"Movies_Genres"."MovieID" =
"Ratings"."MovieID"
GROUP BY "Genres"."GenreName"
ORDER BY "AverageRating" DESC;
```

1) Query Analysis: This query involves aggregating and grouping data to calculate the average ratings for each movie genre. The query utilizes joins between the "Genres," "Movies_Genres," and "Ratings" tables. The result set is then ordered in descending order based on the calculated average ratings.



2) Execution Steps:

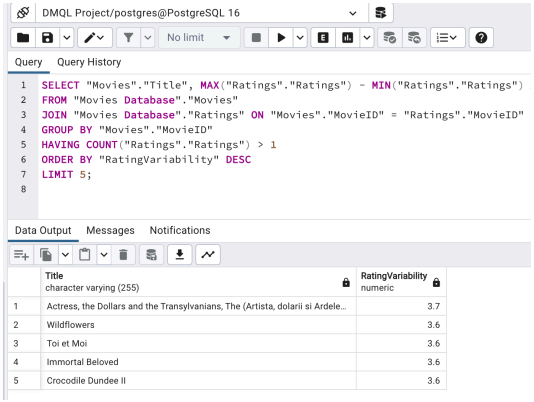
- The database engine performs an inner join between the "Genres" and "Movies_Genres" tables based on the "GenreID" column.
- Another inner join is performed between the result of the first join and the "Ratings" table based on the "MovieID" column.
- The data is grouped by the "GenreName" column from the "Genres" table using the GROUP BY clause.
- For each genre group, the AVG() function calculates the average rating from the "Ratings" table.
- The final result set includes "GenreName" and the corresponding "AverageRating."
- The result set is ordered in descending order based on "AverageRating" using the ORDER BY clause.

3) Query Execution Time : Moderate time required. The query involves multiple joins and aggregation operations, which may introduce moderate processing time, especially with a large number of records. The efficiency is subject to factors like database indexing, table sizes, and server performance.

D) SELECT Query 4 (Using GROUPBY, HAVING, JOIN, ORDERBY)

```
SELECT "Movies"."Title",
MAX("Ratings"."Ratings") -
MIN("Ratings"."Ratings") AS
"RatingVariability"
FROM "Movies Database"."Movies"
JOIN "Movies Database"."Ratings" ON
"Movies"."MovieID" =
"Ratings"."MovieID"
GROUP BY "Movies"."MovieID",
"Movies"."Title"
HAVING COUNT("Ratings"."Ratings") > 1
ORDER BY "RatingVariability" DESC
LIMIT 5;
```

1) Query Analysis: This query calculates the variability in ratings for each movie by finding the difference between the maximum and minimum ratings. It then filters out movies with only one rating using the HAVING clause. The result is ordered by rating variability in descending order, and the top 5 movies with the highest variability are selected.



2) Execution Steps:

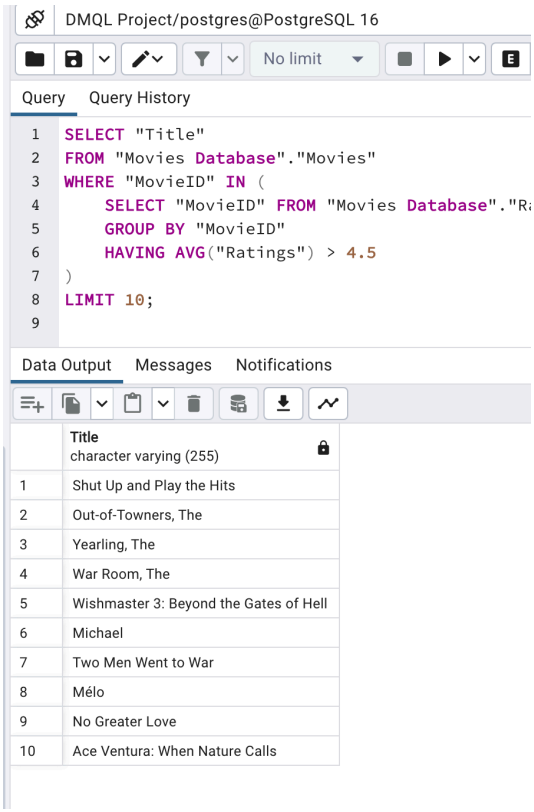
- The database engine performs an inner join between the "Movies" and "Ratings" tables based on matching "MovieID" values.
- For each movie, it calculates the rating variability using MAX(Rating) - MIN(Rating).
- The results are grouped by "MovieID" using the GROUP BY clause.
- The HAVING clause filters out movies with only one rating.
- The result set is then ordered in descending order based on rating variability using ORDER BY.
- Finally, the LIMIT 5 ensures that only the top 5 movies with the highest rating variability are selected.

3) Query Execution Time : Moderate to high time required. The query involves GROUP BY and aggregate functions, potentially requiring processing for movies with a large number of ratings. The HAVING clause adds complexity by filtering out movies with only one rating. Overall, the execution time is influenced by the nature of grouping, aggregation, and filtering operations.

E) SELECT Query 5 (Subquery)

```
  SELECT "Title"
FROM "Movies
Database"."Movies"
WHERE "MovieID" IN (
   SELECT "MovieID"
   FROM "Movies
Database"."Ratings"
   GROUP BY "MovieID"
   HAVING AVG("Rating") > 4.5
);
```

1) *Query Analysis:* The query involves a subquery that calculates the average rating for each movie in the "Ratings" table and a main query that selects the titles of movies with an average rating higher than 4.5. The use of the IN clause for subquery results influences the overall efficiency.



2) *Execution Steps:*

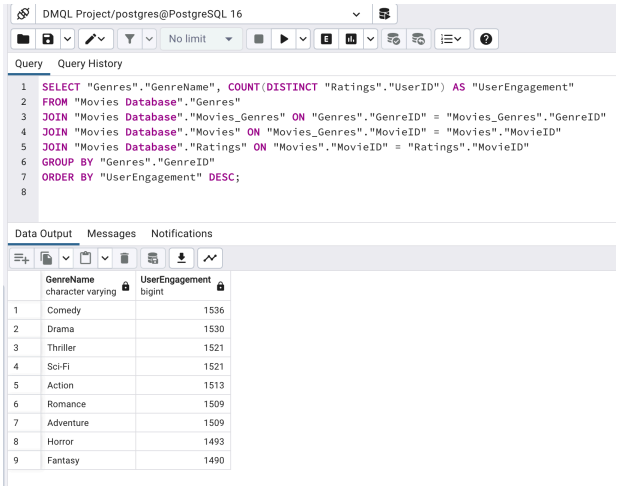
- The subquery scans the "Ratings" table, grouping ratings by movie and calculating the average rating using the AVG() function.
- The HAVING clause filters out movies with an average rating lower than or equal to 4.5.
- The main query selects the titles of movies with MovieIDs satisfying the condition in the subquery.

3) *Query Execution Time:* Moderate. The query involves a subquery with grouping and aggregation operations, and the use of the IN clause for subquery results. While these operations contribute to moderate processing time, the overall complexity is not excessively high, resulting in a moderate execution time.

F) SELECT Query 6 (Using Multiple JOINS, GROUPBY & ORDERBY)

```
SELECT "Genres"."GenreName",
COUNT(DISTINCT
"Ratings"."UserID") AS
"UserEngagement"
FROM "Movies
Database"."Genres"
JOIN "Movies
Database"."Movies_Genres" ON
"Genres"."GenreID" =
"Movies_Genres"."GenreID"
JOIN "Movies Database"."Ratings"
ON "Movies_Genres"."MovieID" =
"Ratings"."MovieID"
GROUP BY
"Genres"."GenreName"
ORDER BY "UserEngagement"
DESC;
```

1) Query Analysis: This query calculates user engagement for each movie genre by counting the distinct users who rated movies in each genre. The result is then ordered by user engagement in descending order.



2) Execution Steps:

- The database engine performs inner joins between the "Genres," "Movies_Genres," and "Ratings" tables based on matching "GenreID" and "MovieID" values.
- For each genre, it counts the distinct users who have rated movies in that genre using

COUNT(DISTINCT "Ratings"."UserID").

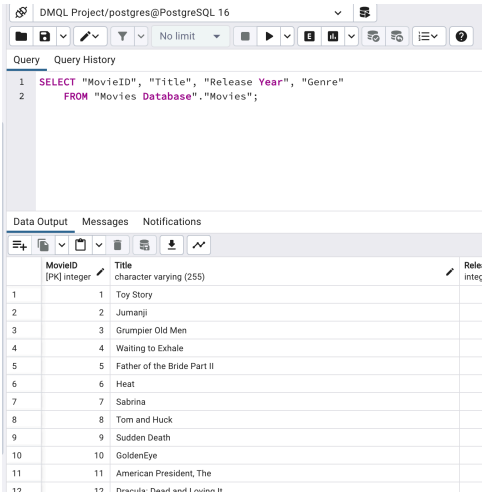
- The results are grouped by "GenreName" using the GROUP BY clause.
- The result set is ordered in descending order based on user engagement using ORDER BY.

3) Query Execution Time : Moderate. The query involves multiple tables joins, grouping, and counting distinct users, which introduces moderate processing complexity. The execution time is influenced by the nature of these operations and the efficiency of the database engine in handling them.

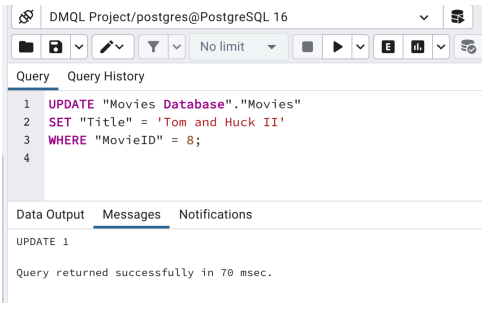
G) UPDATE Query 1

UPDATE"Movies Database"."Movies" SET "Title" = 'Tom and Huck II' WHERE "MovieID" = 8;

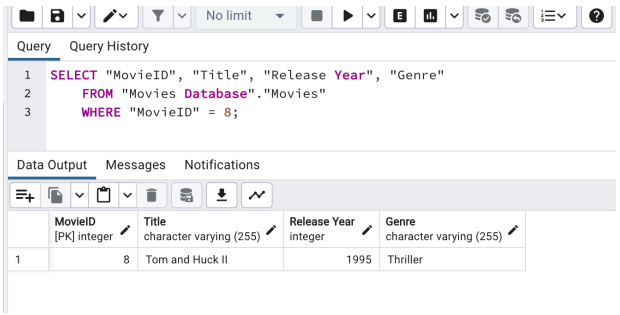
1) Before UPDATE



2) Execute UPDATE



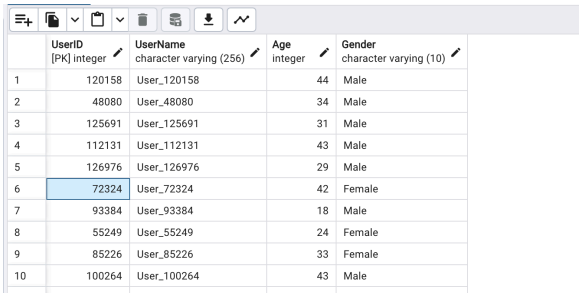
3) After UPDATE



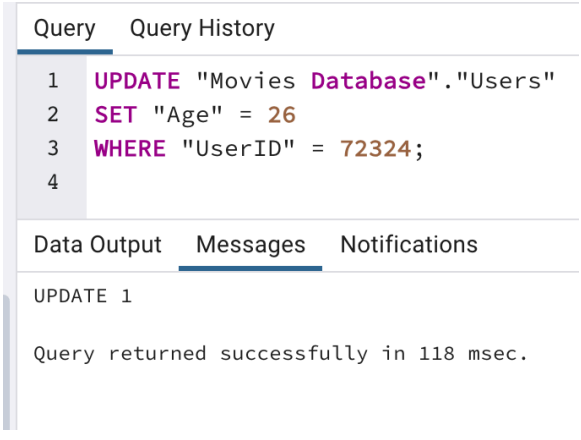
H) UPDATE Query 2

UPDATE "Movies Database"."Users" SET "Age" = 26 WHERE "UserID" = 72324;

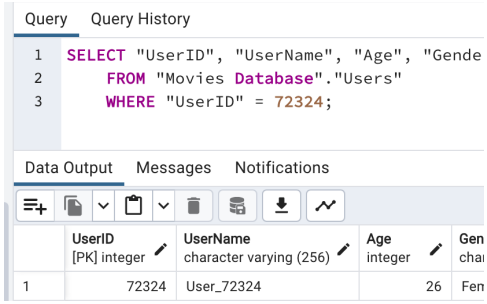
1) Before UPDATE



2) Execute UPDATE

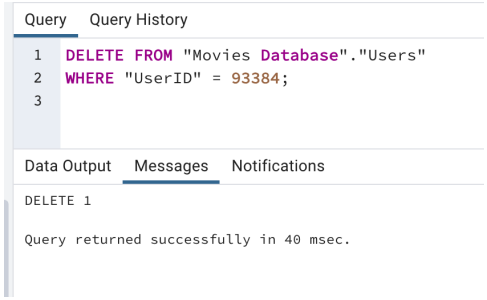


3) After UPDATE

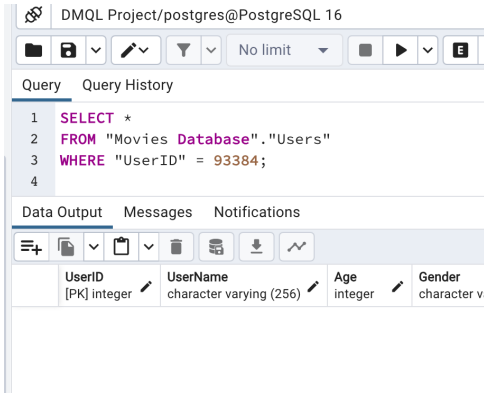


I) DELETE Query 1

DELETE FROM “Movies Database”. “Users”
WHERE “UserID” = 93384;

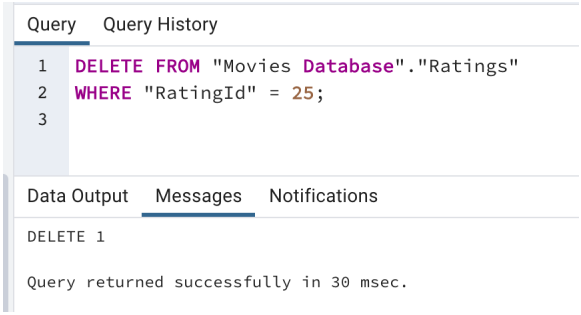


1) Result of Delete Query:

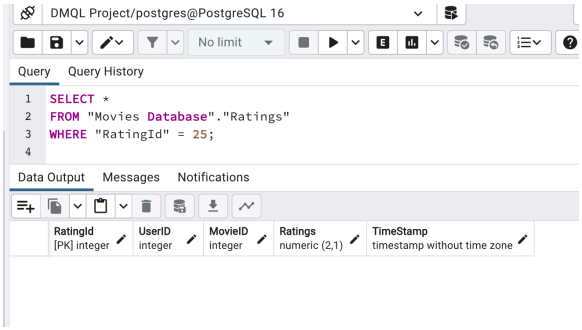


K) DELETE Query 2

DELETE FROM “Movies Database”.”Ratings”
WHERE “RatingId” = 25;



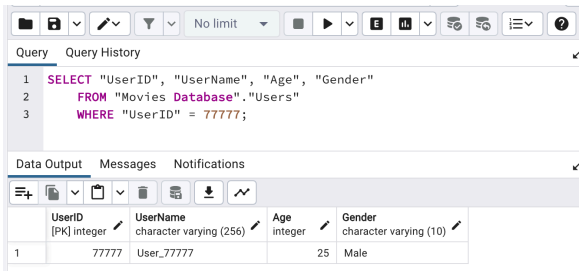
1) Result of Delete Query:



L) INSERT Query 1

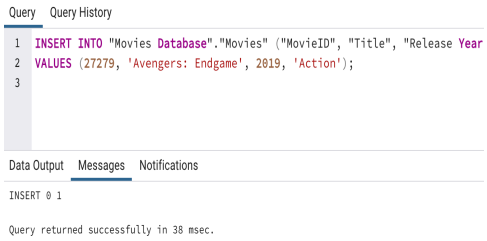
INSERT INTO “Movies Database”.”Users” (“UserID”, “UserName”, “Age”, “Gender”)
VALUES (77777, ‘User_77777’, 25, ‘Male);

1) Result of Insert Query:

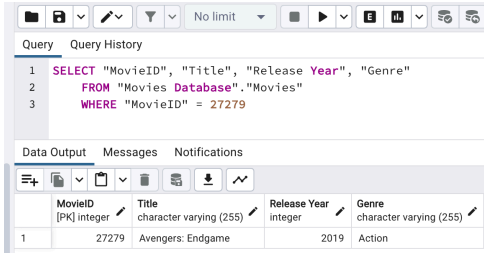


M) INSERT Query 2

INSERT INTO “Movies Database”.”Movies” (“MovieID”, “Title”, “Release Year”, “Genre”) VALUES (27279, “Avengers : Endgame’, 2019, ‘Action’);



1) Result of Insert Query:

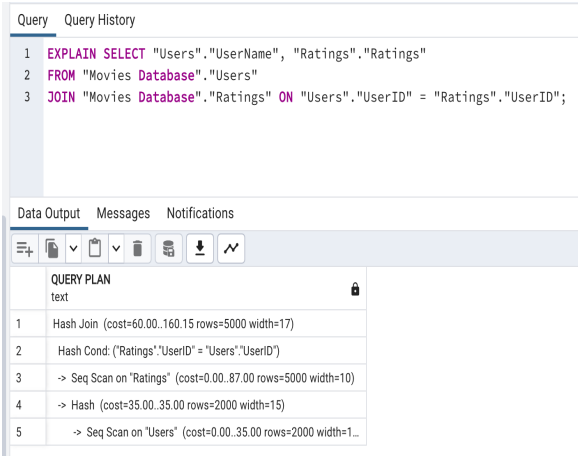


X) QUERY EXECUTION ANALYSIS

A) Query 1 using EXPLAIN :

EXPLAIN SELECT "Users"."UserName", "Ratings"."Rating" FROM "Movies Database"."Users" JOIN "Movies Database"."Ratings" ON "Users"."UserID" = "Ratings"."UserID";

1) *Analysis:* This query involves a basic join between Users and Ratings tables.

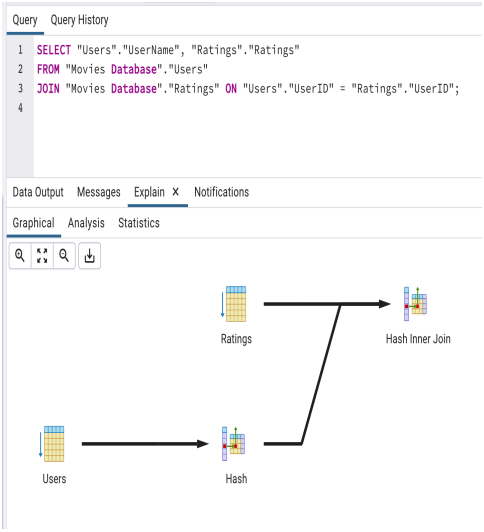


The provided execution plan indicates that the query is performing a Hash Join between the "Ratings" and "Users" tables. Similar to the previous analysis, we can explore potential optimizations to enhance query performance:

2) *Indexing:* Ensure that there is an index on the "Ratings"."UserID" and "Users"."UserID" columns to speed up the Hash Join operation.

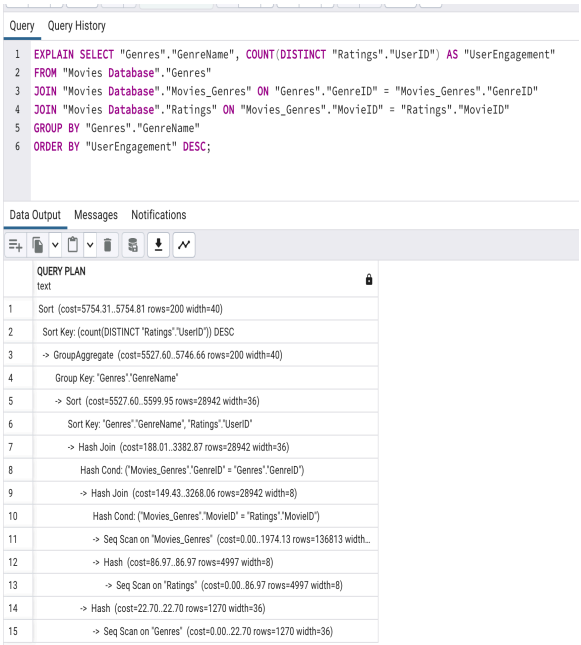
CREATE INDEX idx_ratings_userid ON "Movies Database"."Ratings" ("UserID");
CREATE INDEX idx_users_userid ON "Movies Database"."Users" ("UserID");

3) *Analyze Query Structure:* Check if the query needs to retrieve all rows from the "Ratings" and "Users" tables. If not, consider adding additional conditions to limit the result set.



B) Query 2 using EXPLAIN:-

```
EXPLAIN SELECT
"Genres"."GenreName",
COUNT(DISTINCT
"Ratings"."UserID") AS
"UserEngagement"
FROM "Movies
Database"."Genres"
JOIN "Movies
Database"."Movies_Genres" ON
"Genres"."GenreID" =
"Movies_Genres"."GenreID"
JOIN "Movies Database"."Ratings"
ON "Movies_Genres"."MovieID" =
"Ratings"."MovieID"
GROUP BY
"Genres"."GenreName"
ORDER BY "UserEngagement"
DESC;
```



The resultant **QUERY PLAN** indicates that the query involves multiple operations, including hash joins, sorting, and group aggregation. Here's a breakdown of the key elements:

a) *Hash Joins:*
- There are two hash joins involved in the query:
- The first hash join is between "Movies_Genres" and "Genres" on the "GenreID."
- The second hash join is between "Movies_Genres" and "Ratings" on the "MovieID."

b) *Sorting:*
- There is a sorting operation involved, which is indicated by the "Sort" node.
- The sorting is based on the count of distinct "UserID" in descending order (**Sort Key: (count(DISTINCT "Ratings"."UserID")) DESC**).

c) *Group Aggregation:*
- The query uses the GroupAggregate operation to group the results by "Genres"."GenreName" and perform an

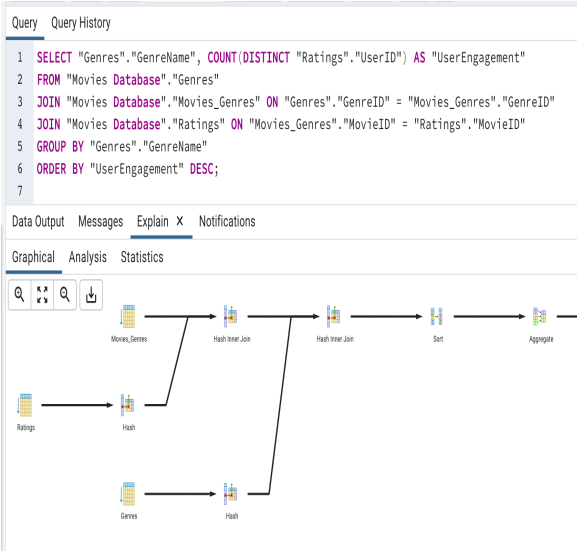
aggregation on the count of distinct "UserID" for each group.

d) Costs:

- The costs associated with the operations are provided, and they represent the estimated resource usage.

1) Optimization Ideas:

- Ensure that there are indexes on columns used in join conditions and WHERE clauses, such as "Movies_Genres"."GenreID," "Movies_Genres"."MovieID," and "Ratings"."UserID."
- Verify that statistics are up-to-date to help the query planner make better decisions.
- Consider indexing on "Ratings"."MovieID" and "Movies_Genres"."GenreID" to speed up the hash joins.



C) Query 3 Using EXPLAIN:

```
EXPLAIN SELECT "Movies"."Title",
MAX("Ratings"."Ratings") -
MIN("Ratings"."Ratings") AS
"RatingVariability"
FROM "Movies Database"."Movies"
JOIN "Movies Database"."Ratings" ON
"Movies"."MovieID" =
"Ratings"."MovieID"
GROUP BY "Movies"."MovieID",
"Movies"."Title"
HAVING COUNT("Ratings"."Ratings") > 1
ORDER BY "RatingVariability" DESC
LIMIT 5;
```

1) Execution Plan Analysis:

a) Filtering and Aggregation:

- The query involves filtering movies with more than one rating and calculating the rating variability.
- The HashAggregate step groups by "MovieID" and applies the count and filter.

b) Hash Join:

- The Hash Join is performed on "Movies" and "Ratings" tables based on "MovieID."

c) Seq Scan:

- Sequential scans are used for both "Movies" and "Ratings" tables, which may lead to performance issues with larger datasets.

2) Improvements:

a) Indexing:

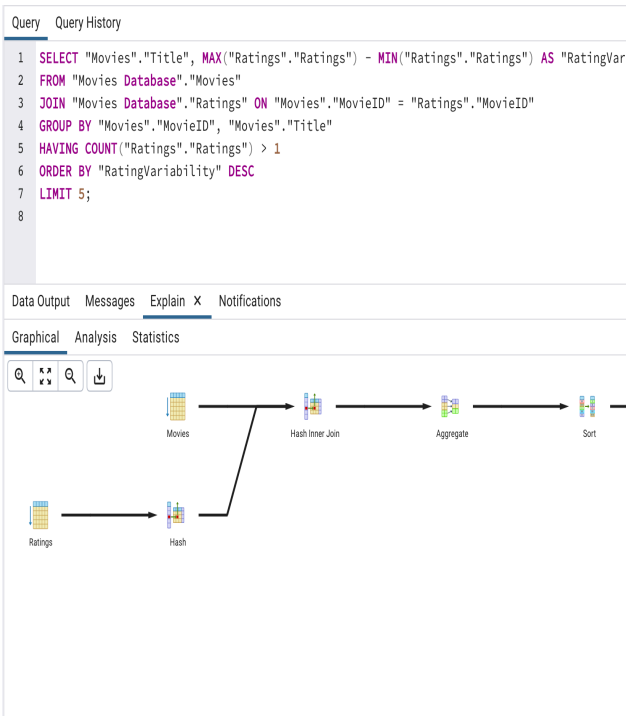
- Ensure that there are indexes on the columns used in join and filter conditions, especially "MovieID" in both "Movies" and "Ratings" tables.
- Ensure that indexes on "MovieID" in both tables are being utilized for join conditions.

b) Optimized Grouping:

- Utilize window functions for more efficient ranking and filtering based on ratings.

c) Reduced Sequential Scans:

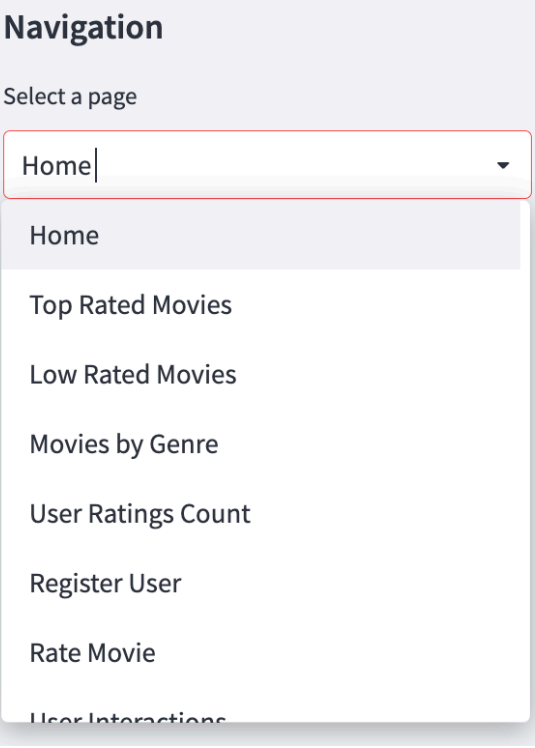
- Minimize sequential scans by optimizing existing indexes or creating new ones.



XI. UI & Its Working

A) Navigation:

- Use the sidebar to navigate between different pages.
- Pages include "Home," "Top Rated Movies," "Low Rated Movies," "Movies by Genre," "User Ratings Count," "Register User," "Rate Movie," and "User Interactions."
- Each page provides specific features and insights related to movie recommendations, ratings, and user interactions.
- The "Refresh Database" button is available on each page to refresh the entire database.



B) UI Home Page :

- Description:
  - The Home Page serves as a central hub, offering both movie recommendations and a comprehensive list of user ratings.
- Movie Recommendations:
  - Recommends 5 random movies along with their titles and genres.

- User Ratings:
    - Displays a table with all movie titles and their corresponding user ratings.
- Tables Used: "Movies" and "Ratings."

- SQL Queries:
    - Recommendations: Randomly selects 5 movies from the "Movies" table.
    - User Ratings: Retrieves all movies and their ratings from the "Movies" and "Ratings" tables.

C) Top Rated Movies Page:

- Description:
    - The Top Rated Movies Page showcases a list of top-rated movies based on average user ratings.
- Top Rated Movies:
    - Displays a table with movie titles and their average ratings.
- Tables Used:
    - "Movies" and "Ratings."
- SQL Queries:
    - Retrieves the top-rated movies by calculating the average rating for each movie.

D) Low Rated Movies Page :

- Description:
    - The Low Rated Movies Page presents a list of movies with lower average ratings.
- Low Rated Movies:
    - Shows a table with movie titles and their below-average ratings.
- Tables Used:
    - "Movies" and "Ratings."
- SQL Queries:
    - Retrieves low-rated movies with an average rating of 2.5 or below.

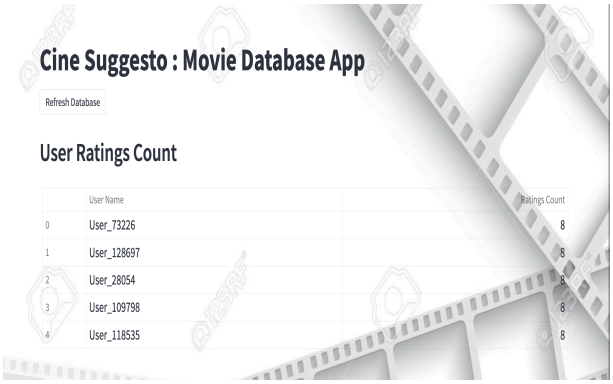
E) Movies By Genre Page :

- Description:
    - The Movies by Genre Page allows users to explore movies based on different genres.
- Genre Selection:

- Users can select a genre and view related movies.
- Tables Used:
  - "Movies," "Movies_Genres," and "Genres."
- SQL Queries:
  - Retrieves movies based on the selected genre from the "Movies," "Movies_Genres," and "Genres" tables.

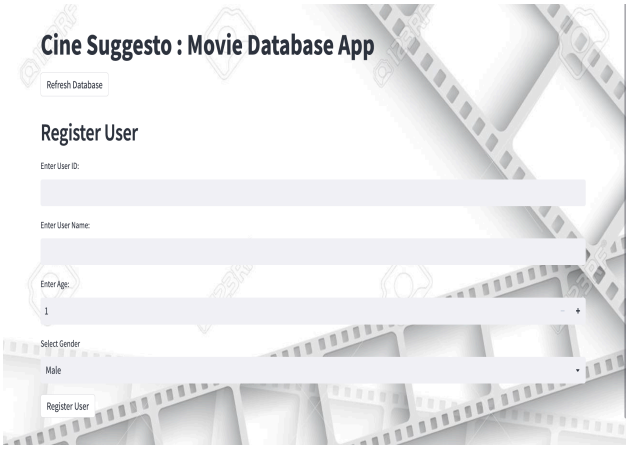
F) Top User Ratings Count Page :

- Description:
  - The User Ratings Count Page provides the count of ratings given by users.
- Ratings Count:
  - Displays a table with user names and their corresponding ratings count.
- Tables Used:
  - "Users" and "Ratings."
- SQL Queries:
  - Retrieves user ratings count from the "Users" and "Ratings" tables



Register User Page :

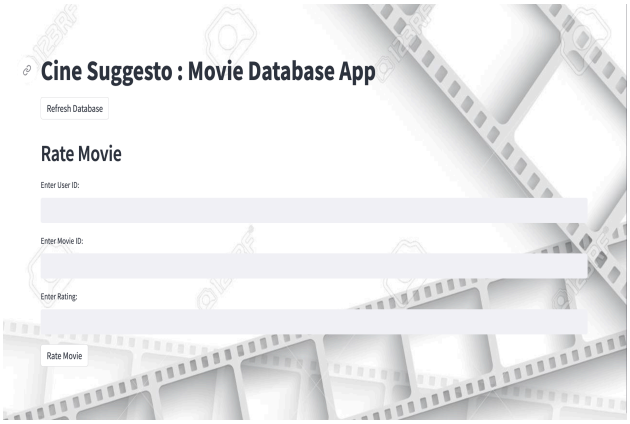
- Description:
  - The Register User Page allows users to register by providing necessary details.
- User Registration:
  - Users can enter a unique ID, name, age, and gender to register.
- Tables Used:
  - "Users."
- SQL Queries:
  - Inserts a new user record into the "Users" table.



G) Rate Movie Page :

- Description:
  - The Rate Movie Page enables users to rate a movie by providing user and movie details.
- Movie Rating:

- Users can enter their ID, movie ID, and a rating to submit.
- Tables Used:
  - "Ratings" and "Users."
- SQL Queries:
  - Inserts a new rating record into the "Ratings" table.



H) User Interactions Page :

- Description:
  - The User Interactions Page allows users to retrieve their rated movies and view sample user IDs.
- Retrieve Ratings:
  - Users can input their ID to see a table of rated movies.
- Sample User IDs:
  - Shows sample user IDs for reference.
- Tables Used:
  - "Movies," "Ratings," and "Users."
- SQL Queries:
  - Retrieves user ratings based on the provided user ID from the "Movies" and "Ratings" tables.
  - Shows sample user IDs from the "Users" table.