Enhancing Viewer Engagement Through Data-Driven Content Personalization

Amulya Reddy Datla
UBIT Name: amulyare
UB Person Number: 50560100
email: amulyare@buffalo.edu

Pratiksha Rajendra Pawar
UBIT Name: ppawar2
UB Person Number: 50560590
email: ppawar2@buffalo.edu

I. Problem Statement

In the competitive world of streaming, enhancing viewer engagement through personalized content is key to reducing churn and increasing watch time. This project seeks to leverage Netflix's extensive dataset to uncover how factors such as genres, cast, and directors influence viewer preferences. By identifying the personalization drivers that resonate most with different audience segments, we aim to deliver actionable insights for improving Netflix's recommendation algorithms, ultimately creating a more tailored and engaging user experience.

II. database over an Excel file

The below mentioned are the reasons for opting a database over an Excel file

- Databases can handle much larger datasets than Excel, making them ideal for scaling.
- They maintain data integrity by enforcing relationships and rules, reducing errors.
- Multiple users can access and update a database simultaneously without conflicts.
- Databases allow for advanced querying and automation, which Excel struggles with.
- They offer stronger security features and reliable backup options compared to Excel.

III. Target Users

A. Users of Database

1) Netflix Data Scientists: These professionals will use the database to conduct in-depth analysis of viewer preferences, running machine learning models to predict engagement and fine-tune recommendation algorithms based on factors such as genres, directors, and cast.

2) Content Curators and Editors: Responsible for selecting and promoting content, curators will use insights from the database to tailor recommendations to specific audience segments, ensuring that content resonates with users based on their watching habits.

3) Marketing Teams: Marketers will leverage the database to craft targeted campaigns, utilizing viewer data to promote specific shows or movies to users most likely to engage, ultimately boosting watch time and reducing churn.

4) Streaming Engineers and Data Engineers: These professionals will use the data to maintain and improve the streaming platform's recommendation engine, ensuring its efficiency and relevance while managing the technical aspects of the database.

5) Film and Media Researchers: Academic researchers can utilize the dataset to study the trends in audience preferences, examining the impact of personalization factors such as genres or cast on viewer engagement and developing theories related to media consumption behavior.

6) Business Analysts and Executives: Key decision-makers will use high-level insights from the database to guide strategic content investment, understanding which types of content drive the highest engagement across various viewer segments.

7) Database Administrators: Responsible for managing, maintaining, and securing the database, these individuals ensure its integrity, performance, and availability. They handle backups, security, and scaling to ensure the database runs efficiently for all users.

B. Administrator of the Database

Database Administrator play a crucial role in maintaining the integrity and performance of the database, especially during processes like updating recommendations and launching marketing campaigns.

C. Real Life Scenario

Netflix notices declining viewer engagement in a region and uses its database to analyze user preferences, finding that viewers favor action-thriller content with local actors. With this insight, the database administrator updates the recommendations and launches targeted marketing campaigns. As a result, engagement increases, churn decreases, and the database administrators ensure everything runs smoothly during the process.

IV. Data Transformation

The ipynb notebook analyzes a Netflix dataset by cleaning the data—removing null values and duplicates, renaming columns like 'listed in' to 'category' and 'cast' to 'actor', and trimming entries to keep only the primary actor and category. It visualizes content distribution, top producing countries, and trends in content addition over time. The analysis also explores relationships between content duration and release year, concluding with the export of the processed data to a new CSV file.

V. E/R Diagram

The Entity-Relationship (ER) diagram illustrates the relationships between different entities in a database system. This diagram is crucial for understanding how data is structured and interconnected. It helps in designing a database by clearly defining the tables, their attributes, and the relationships among them. Shows, Directors, Actors, Countries, and Categories. Each table has a primary key (PK) for unique identification and may have foreign keys (FK) to establish relationships with other tables.

A. Keys and Attributes

1) Shows:

- Primary Key: 'show id'
  The "show id" serves as a unique identifier for each record in the Shows table. It ensures that each show entry is distinct and can be referenced unambiguously in queries and operations.
- Foreign Keys:
  'director id' references Directors - Links each show to a director, allowing multiple shows to be associated with one director.
  'actor id' references Actors - Connects shows to actors, enabling multiple shows to feature the same actor.
  'country id' references Countries - Associates shows with their country of origin, allowing many shows to be produced in one country.
  'category id' references Category - Categorizes shows, permitting multiple shows to fall under the same category.
- Attributes:
  show id - Integer, Primary Key, NOT NULL
  director id - Integer, Foreign Key
  director - String
  type - String
  actor id - Integer, Foreign Key
  actor - String
  title - String, NOT NULL
  date added - Date
  release year - Integer
  country id - Integer, Foreign Key
  country - String
  rating - String
  duration - String
  description - Text
  category id - Integer, Foreign Key
  category - String
- Relationship: Many-to-One (Many shows can be directed by one director, Many shows can feature one actor, Many shows can be produced in one country, Many shows can belong to one category.)

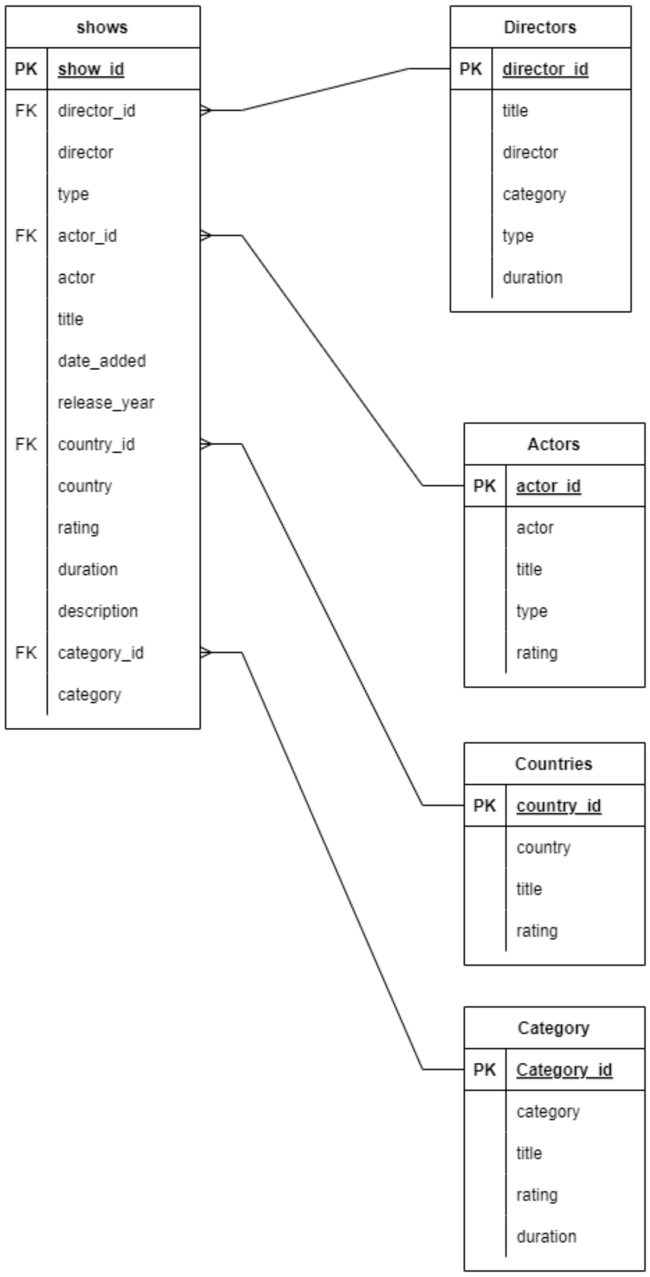


Fig. 1. Entity-Relationship Diagram

2) Directors:

- Primary Key: 'director id'
  The director id uniquely identifies each director in the Directors table, ensuring that each entry is distinct.
- Attributes:
  director id - Integer, Primary Key, NOT NULL
  title - String
  director - String, NOT NULL
  category - String
  type - String
  duration - String

- Relationship: One-to-Many with Shows (a director can direct multiple shows)

3) Actors:

- Primary Key: 'actor id'
- Attributes:
  actor id - Integer, Primary Key, NOT NULL
  actor - String, NOT NULL
  title - String
  type - String
  rating - String
- Relationship: One-to-Many with Shows (an actor can appear in multiple shows)

4) Countries:

- Primary Key: 'country id'
- Attributes:
  country id - Integer, Primary Key, NOT NULL
  country - String, NOT NULL
  title - String
  rating - String
- Relationship: One-to-Many with Shows (In a country multiple shows can be shooted)

5) Category:

- Primary Key: 'category id'
- Attributes:
  category id - Integer, Primary Key, NOT NULL
  category - String, NOT NULL
  title - String
  rating - String
  duration - String
- Relationship: One-to-Many with Shows (a category can include multiple shows)

B. Description of Attributes

1) Shows Table:

- show id: Integer. Identifier for the show.
- director id: Integer, Foreign Key. References director id in Directors.
- director: String. Name of the director.
- type: String. Type or genre of the show.
- actor id: Integer, Foreign Key. References actor id in Actors.
- actor: String. Name of the actor.
- title: String. Title of the show.
- date added: Date. Date when the show was added to the database.
- release year: Integer. Year the show was released.
- country id: Integer, Foreign Key. References country id in Countries.
- country: String. Country where the show was produced.
- rating: String. Rating of the show (e.g., PG, R).
- duration: String. Duration of the show (e.g., 90 min).
- description: Text. Brief description or synopsis of the show.
- category id: Integer, Foreign Key. References Category id in Category.
- category: String. Category or genre of the show.

2) Directors Table:

- director id: Integer, Primary Key. Unique identifier for each director.
- show id: Integer, Foreign Key. References id in Shows.
- title: String. Title of directed shows.
- director: String. Name of the director.
- category: String. Category or genre of directed shows.
- type: String. Type or genre of directed shows.
- duration: String. Duration of directed shows.

3) Actors Table:

- actor id: Integer, Primary Key. Unique identifier for each actor.
- actor: String. Name of the actor.
- title: String. Title of acted shows.
- type: String. Type or genre of acted shows.
- rating: String. Rating of acted shows.

4) Countries Table:

- country id: Integer, Primary Key. Unique identifier for each country.
- country: String. Name of the country.
- title: String. Title associated with the country's shows.
- rating: String. Rating associated with the country's shows.

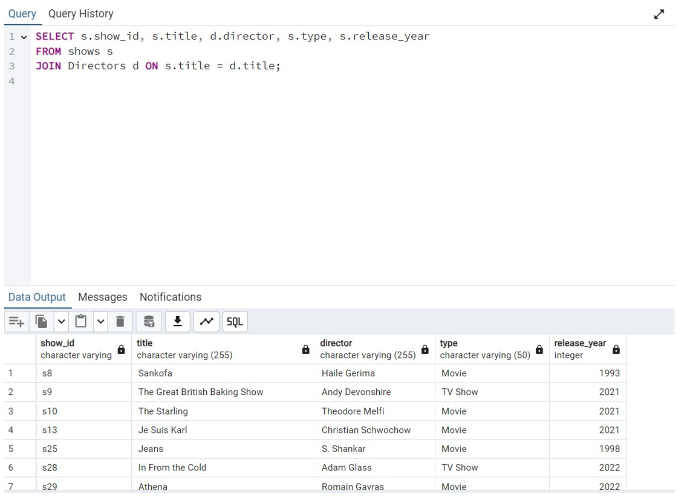
5) Category Table:

- Category id: Integer, Primary Key. Unique identifier for each category.
- category: String. Name of the category or genre.
- title: String. Title associated with the category's shows.
- rating: String. Rating associated with the category's shows.
- duration: String. Duration associated with the category's shows.

The ER diagrams depict the relationships between the Shows, Directors, Actors, Countries, and Category tables. The Shows table uses id as its primary key and includes foreign keys such as director id, actor id, country id, and category id, which reference the primary keys in the Directors, Actors, Countries, and Category tables, respectively. This setup establishes many-to-one relationships where multiple shows can be associated with a single director, actor, country, or category. The Directors table has director id as its primary key and includes a foreign key show id referencing the Shows table, indicating that a director can be linked to multiple shows. Each table's attributes are designed to capture essential information about the entities they represent, ensuring efficient data organization and retrieval within the database system.

C. SQL Queries

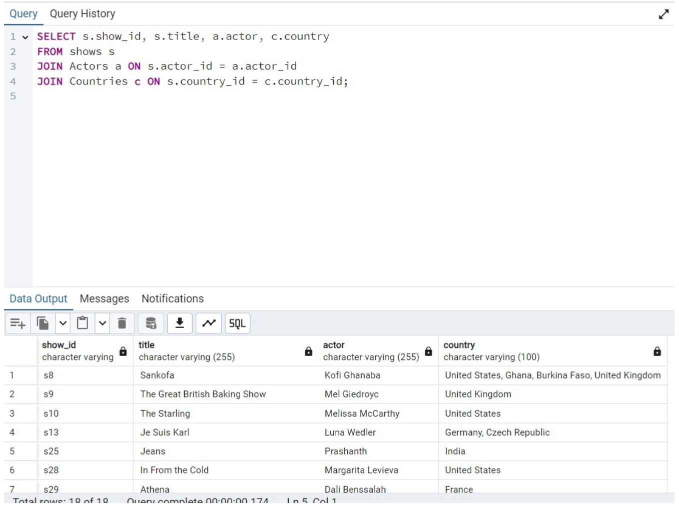
The SQL queries demonstrate a range of database operations. These queries illustrate common database tasks such as data retrieval, modification, deletion, and filtering, showcasing capability of SQL to manage and analyze relational data effectively. SQL queries are powerful tools for interacting with relational databases, allowing users to retrieve, manipulate, and manage data efficiently. Based on the database schema shown, deleting a show with show_id=s8 would have no cascading effects on other attributes and tables because shows table contains foreign keys that reference other tables (Directors, Actors, Countries, Category) but is not referenced by any other tables as a foreign key target. If a primary key is deleted in a database, the foreign keys that reference it are affected based on the referential actions defined during the foreign key constraint setup. All these queries presented below showcase the versatility of SQL in retrieving, updating, deleting, and filtering data across multiple tables in a relational database, highlighting its power in managing and analyzing complex datasets.



```
SELECT s.show_id, s.title, d.director, s.type, s.release_year
FROM shows s
JOIN Directors d ON s.title = d.title;
```

	show_id character varying	title character varying (255)	director character varying (255)	type character varying (50)	release_year integer
1	s8	Sankofa	Haile Gerima	Movie	1993
2	s9	The Great British Baking Show	Andy Devonshire	TV Show	2021
3	s10	The Starling	Theodore Melfi	Movie	2021
4	s13	Je Suis Karl	Christian Schwochow	Movie	2021
5	s25	Jeans	S. Shankar	Movie	1998
6	s28	In From the Cold	Adam Glass	TV Show	2022
7	s29	Athena	Romain Gavras	Movie	2022

Fig. 2. : Query -1 :The above query select all shows with director information (joining shows and directors tables)

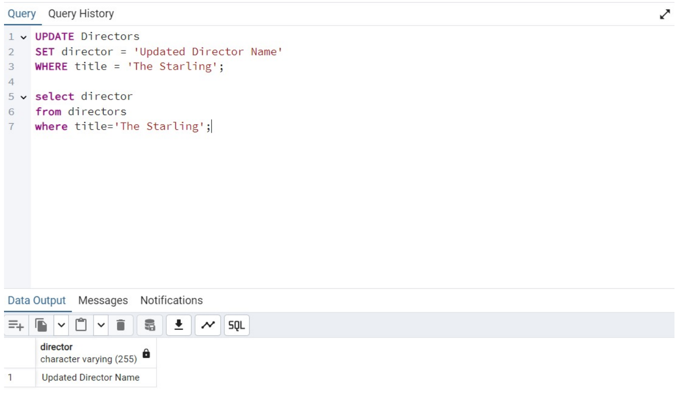


```
SELECT s.show_id, s.title, a.actor, c.country
FROM shows s
JOIN Actors a ON s.actor_id = a.actor_id
JOIN Countries c ON s.country_id = c.country_id;
```

	show_id character varying	title character varying (255)	actor character varying (255)	country character varying (100)
1	s8	Sankofa	Kofi Ghanaba	United States, Ghana, Burkina Faso, United Kingdom
2	s9	The Great British Baking Show	Mel Giedroyc	United Kingdom
3	s10	The Starling	Melissa McCarthy	United States
4	s13	Je Suis Karl	Luna Wedler	Germany, Czech Republic
5	s25	Jeans	Prashanth	India
6	s28	In From the Cold	Margarita Levieva	United States
7	s29	Athena	Dali Benssalah	France

```
Total rows: 18 of 18     Query complete 00:00:00.174     Ln 5, Col 1
```

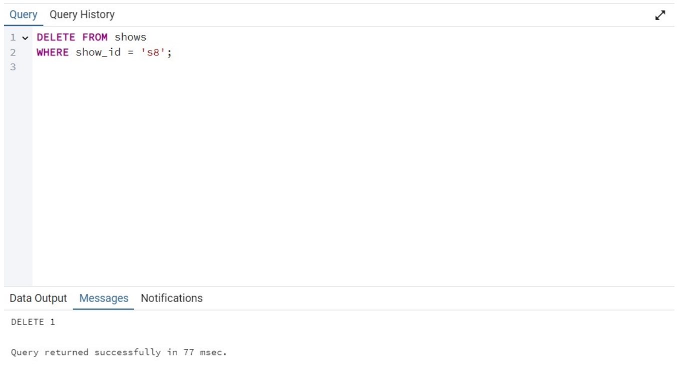
Fig. 3. Query - 2: The above query select all shows along with their main actor and country information



```
UPDATE Directors
SET director = 'Updated Director Name'
WHERE title = 'The Starling';

select director
from directors
where title='The Starling';
```

	director character varying (255)
1	Updated Director Name

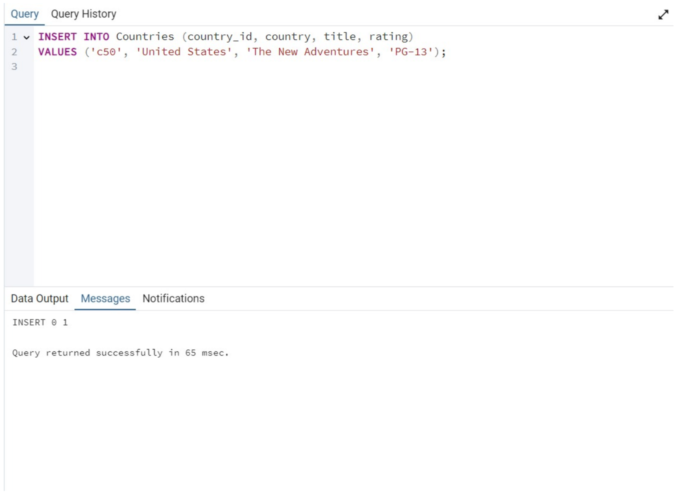
Fig. 4. : Query - 3 : The above query Update the director of a specific show



```
DELETE FROM shows
WHERE show_id = 's8';
```

```
DELETE 1

Query returned successfully in 77 msec.
```

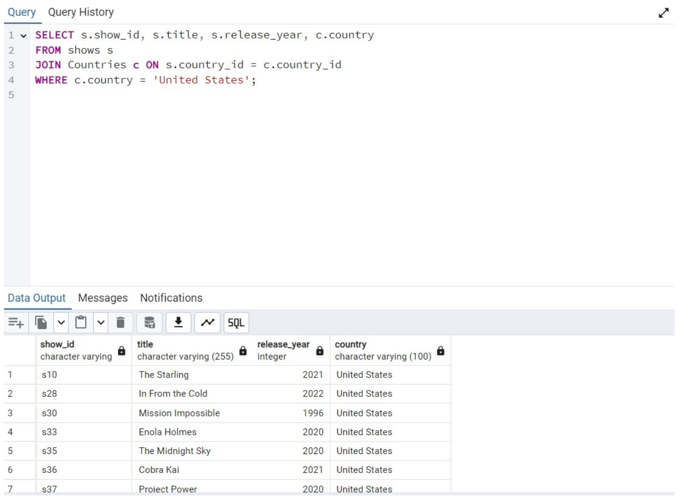
Fig. 5. : Query - 4: The above query deletes the show in the shows table



```
INSERT INTO Countries (country_id, country, title, rating)
VALUES ('c50', 'United States', 'The New Adventures', 'PG-13');
```

```
INSERT 0 1

Query returned successfully in 65 msec.
```

Fig. 6. : Query - 5 : The above query inserts the row in country column

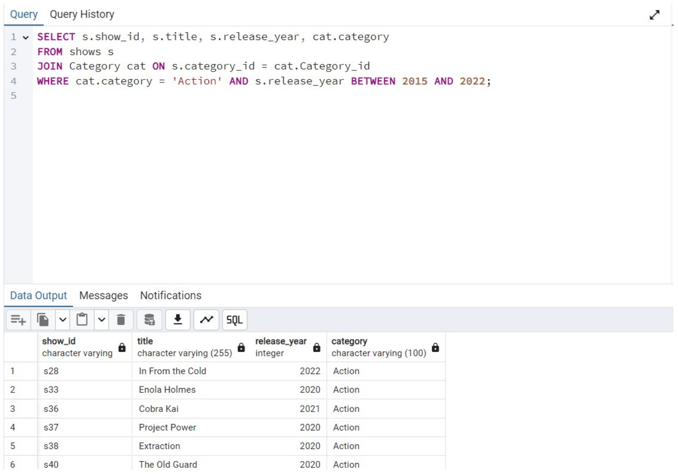

Query Query History

```
SELECT s.show_id, s.title, s.release_year, c.country
FROM shows s
JOIN Countries c ON s.country_id = c.country_id
WHERE c.country = 'United States';
```

Data Output Messages Notifications

	show_id character varying	title character varying (255)	release_year integer	country character varying (100)
1	s10	The Starling	2021	United States
2	s28	In From the Cold	2022	United States
3	s30	Mission Impossible	1996	United States
4	s33	Enola Holmes	2020	United States
5	s35	The Midnight Sky	2020	United States
6	s36	Cobra Kai	2021	United States
7	s37	Project Power	2020	United States

Fig. 7. : Query - 6: The above query returns all shows that are from the United States


Query Query History

```
SELECT s.show_id, s.title, s.release_year, cat.category
FROM shows s
JOIN Category cat ON s.category_id = cat.Category_id
WHERE cat.category = 'Action' AND s.release_year BETWEEN 2015 AND 2022;
```

Data Output Messages Notifications

	show_id character varying	title character varying (255)	release_year integer	category character varying (100)
1	s28	In From the Cold	2022	Action
2	s33	Enola Holmes	2020	Action
3	s36	Cobra Kai	2021	Action
4	s37	Project Power	2020	Action
5	s38	Extraction	2020	Action
6	s40	The Old Guard	2020	Action

Fig. 8. : Query - 7: The above query returns shows that belong to the Action category and were released between 2015 and 2022.

VI. Functional dependencies

Functional dependencies in a relational database design are used to represent the relationship between attributes in a table. Functional dependency indicates that the value of one set of attributes uniquely determines the value of another set of attributes. Functional dependencies are essential for database normalization, as they help identification of the primary key, minimize redundancy, and ensure data integrity by eliminating anomalies such as update or insertion issues.

Boyce-Codd Normal Form (BCNF) and Third Normal Form (3NF) are database normalization forms designed to reduce redundancy and prevent dependency anomalies. A non-trivial functional dependency is one where Y is not a subset of X in X→Y. A relation is in BCNF if, for every non-trivial functional dependency X→Y, the determinant X must be a superkey. This means the attribute or set of attributes on the left side of the functional dependency must uniquely identify every tuple in the relation. An attribute that is part of a candidate key is considered a prime attribute. A relation is in 3NF if, for every functional dependency X→Y, either X is a superkey or Y is a prime attribute. A relation that satisfies BCNF will also satisfy the conditions for 3NF, but the reverse is not necessarily true. A relation can be in 3NF without being in BCNF.

A. *Functional dependendencies and their analysis*

We have considered the following functional dependencies for the relational database design. The database schema comprises five relations namely "Shows", "Directors", "Actors", "Countries", and "Category". The functional dependencies for each relation are outlined below.

1) Shows:

- Shows table consists of attributes show_id, director_id, director, type, actor_id, actor, title, date_added, release_year, country_id, country, rating, duration, description, category_id, category.
- Primary key is show_id.
- The functional dependencies for Shows table are as below
  - show_id → {director_id, actor_id, title, date_added, release_year, country_id, rating, duration, description, category_id}
    - ∗ The primary key show_id is capable of uniquely identifying all the attributes in the set of attributes on the right-hand side of the dependency. That means show_id can determine all other attributes in the relation. In terms of BCNF, for a functional dependency X→Y, X must be a superkey. Since show_id is the primary key and determines all the other attributes, the determinant (show_id) is considered a superkey.
    - ∗ The above functional dependency is in BCNF.
  - director_id → {title, director, category, type, duration}
    - ∗ director_id only uniquely identifies title, director, category, type and duration but it does not uniquely identify an entire tuple in the Shows table. The show_id would be needed to uniquely identify a record, but director_id by itself does not satisfy this criterion. This functional dependency violates BCNF because director_id is not a superkey.
    - ∗ The above functional dependency is not in BCNF.
  - actor_id → {actor, title, type, rating}
    - ∗ actor_id only uniquely identifies actor, title, type and rating but it does not uniquely identify an entire tuple in the Shows table. The show_id would be needed to uniquely identify a record, but actor_id by itself does not satisfy this criterion. This functional dependency violates BCNF because actor_id is not a superkey.

    - ∗ The above functional dependency is not in BCNF.
  - country_id $\rightarrow$ {country, title, rating}
    - ∗ country_id only uniquely identifies a country, title nd rating but it does not uniquely identify an entire tuple in the Shows table. The show_id would be needed to uniquely identify a record, but country_id by itself does not satisfy this criterion. This functional dependency violates BCNF because country_id is not a superkey.
    - ∗ The above functional dependency is not in BCNF.
  - category_id $\rightarrow$ {category, title, rating, duration}
    - ∗ category_id only uniquely identifies a category, title, rating and duration but it does not uniquely identify an entire tuple in the Shows table. The show_id would be needed to uniquely identify a record, but category_id by itself does not satisfy this criterion. This functional dependency violates BCNF because category_id is not a superkey.
    - ∗ The above functional dependency is not in BCNF.
- The Shows relation is not in BCNF due to the presence of few functional dependencies that are not in BCNF as they have non-superkey determinants. To make the schema BCNF-compliant, the Shows relation can be decomposed into smaller relations, as described in the earlier analysis. Shows table can be decomposed to four smaller tables namely Directors, Actors, Countries, Category. This decomposition is done earlier and the below tables are obtained.

2) Directors:

- Directors table consists of attributes director_id, title, director, category, type, duration.
- Primary key is director_id.
- The functional dependencies for Directors table are as below
  - director_id $\rightarrow$ title, director, category, type, duration
  - The functional dependencies obtained by splitting the original functional dependency of the Directors table are director_id $\rightarrow$ director, director_id $\rightarrow$ title, director_id $\rightarrow$ category, director_id $\rightarrow$ type and director_id $\rightarrow$ duration. All these functional dependencies have the determinant as director_id which is a superkey and so all the FDs satisfy the condition for BCNF.
- The Directors relation is in BCNF as all the functional dependencies in the Directors table are in BCNF.

3) Actors:

- Actors table consists of attributes actor_id, actor, title, type, rating.
- Primary key is actor_id.
- The functional dependencies for Actors table are as below
  - actor_id $\rightarrow$ actor, title, type, rating
  - The functional dependencies obtained by splitting the original functional dependency of the Actors table are actor_id $\rightarrow$ actor, actor_id $\rightarrow$ title, actor_id $\rightarrow$ type and actor_id $\rightarrow$ rating. All these functional dependencies have the determinant as actor_id which is a superkey and so all the FDs satisfy the condition for BCNF.
- The Actors relation is in BCNF as all the functional dependencies in the Actors table are in BCNF.

4) Countries:

- Countries table consists of attributes country_id, country, title, rating.
- Primary key is country_id.
- The functional dependencies for Countries table are as below
  - country_id $\rightarrow$ country, title, rating
  - The functional dependencies obtained by splitting the original functional dependency of the Country table are country_id $\rightarrow$ country, country_id $\rightarrow$ title and country_id $\rightarrow$ rating. All these functional dependencies have the determinant as country_id which is a superkey and so all the FDs satisfy the condition for BCNF.
- The Countries relation is in BCNF as all the functional dependencies in the Countries table are in BCNF.

5) Category:

- Category table consists of attributes category_id, category, title, rating, duration.
- Primary key is category_id.
- The functional dependencies for Category table are as below
  - category_id $\rightarrow$ category, title, rating, duration
  - The functional dependencies obtained by splitting the original functional dependency of the Category table are category_id $\rightarrow$ category, category_id $\rightarrow$ title, category_id $\rightarrow$ rating and category_id $\rightarrow$ duration. All these functional dependencies have the determinant as category_id which is a superkey and so all the FDs satisfy the condition for BCNF.
- The Category relation is in BCNF as all the functional dependencies in the Category table are in BCNF.

Now all the relations are in BCNF namely “Directors”, “Actors”, “Countries”, and “Category” which are decomposed from “Shows” relation

VII. Finalized E/R diagram and Constraints

A. Finalized E/R diagram

The finalized Entity relationship diagram is as below. The original Shows table has five functional dependencies in which only one satisfies the BCNF conditions. Decomposition is done accordingly to form the new relations which satisfy the BCNF conditions. In this process tables Directors, Actors, Countries, Category are obtained and the functional dependencies concerned with these tables satisfy the BCNF conditions. All these tables are in BCNF. The finalized E/R diagram is as below.

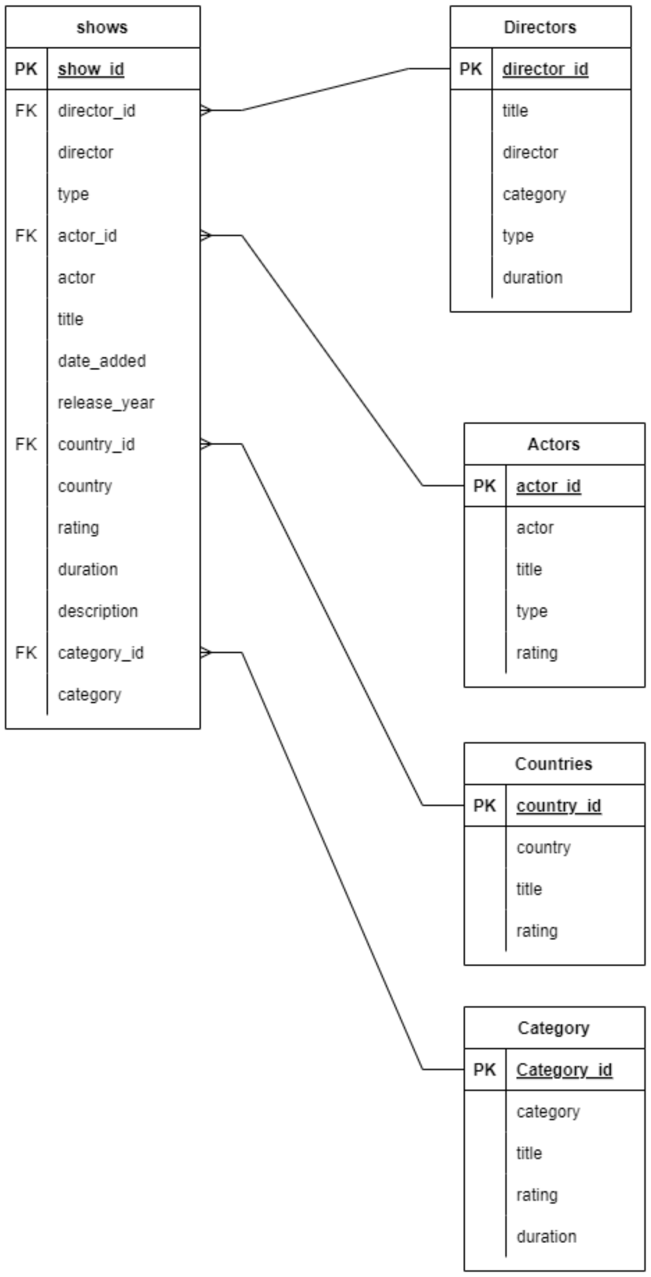


Fig. 9. Entity-Relationship Diagram

B. Constraints

Constraints that can be applicable in our usecase problem are as below

- Primary Key Constraints
  Every table should have a primary key which is an unique identifier to ensure that each record is uniquely identifiable. show_id in the Shows table ensures each show has a unique identifier. director_id in the Directors table uniquely identifies each director. actor_id in the Actors table uniquely identifies each actor. country_id in the Countries table uniquely identifies each country. category_id in the Category table uniquely identifies each category. This constraint ensures every record in the table is uniquely identifiable.

- Foreign Key Constraint
  This constraint establishes a relationship between two tables, ensuring referential integrity. director_id in the Shows table references director_id in the Directors table. actor_id in the Shows table references actor_id in the Actors table. country_id in the Shows table references country_id in the Countries table. category_id in the Shows table references category_id in the Category table. If a referenced record is deleted, dependent records in the Shows table are also deleted. If the referenced key of the record changes, it updates the corresponding foreign key in the Shows table.

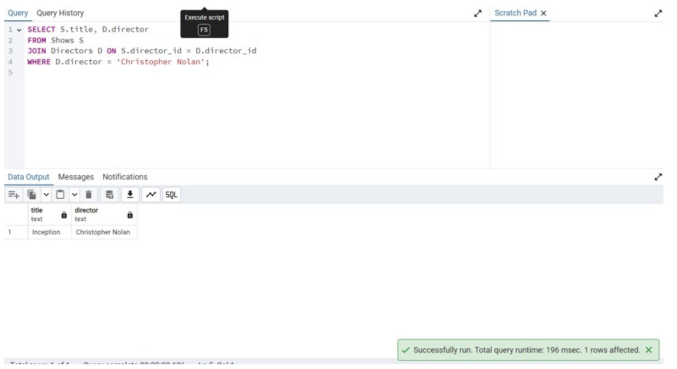
- Check Constraints
  Check Constraints define rules that column values must follow to ensure data validity. For instance, the type column in the Shows table must only include 'Movie' or 'TV Show', and the release year should not exceed the current year to prevent future dates. This constraint enforces data integrity by restricting invalid entries.

- Not Null Constraint
  This constraint ensures that critical columns always have a value and do not allow nulls. The title in Shows ensures every show has a name, while date_added records when it was added. Country in Countries and category in Category ensure proper naming, while actor in Actors and title in Directors ensure actors and directors' works are named, maintaining data accuracy.

VIII. Challenges with Large Datasets and Solutions

A. Challenges faced while handling the larger dataset

- Querying across multiple tables
  Working with large datasets, joins between tables like Shows and other tables can become slow due to the involvement of extensive data. To optimize performance, creating indexes on foreign key columns is done to significantly speed up join operations. For instance, adding indexes on columns like director_id, actor_id, country_id, and category_id in the Shows table ensures faster lookups and more efficient joins, thus reducing query execution time. This approach helps maintain system responsiveness even as datasets grow larger. The difference in the execution time with and without indexing can be clearly visualized in the below images.

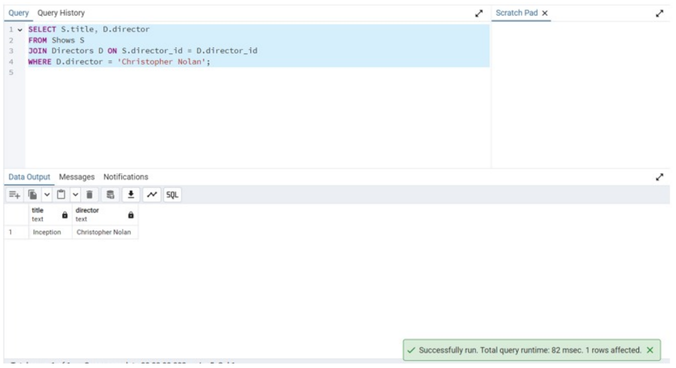


```
SELECT S.title, D.director
FROM Shows S
JOIN Directors D ON S.director_id = D.director_id
WHERE D.director = 'Christopher Nolan';
```

	title text	director text
1	Inception	Christopher Nolan

Successfully run. Total query runtime: 196 msec. 1 rows affected.

Fig. 10. The above query is performed without indexing and the runtime is 196 milli seconds



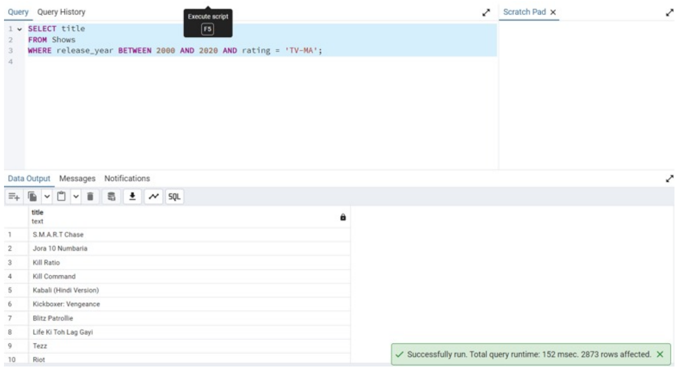
```
SELECT S.title, D.director
FROM Shows S
JOIN Directors D ON S.director_id = D.director_id
WHERE D.director = 'Christopher Nolan';
```

	title text	director text
1	Inception	Christopher Nolan

Successfully run. Total query runtime: 82 msec. 1 rows affected.

Fig. 11. The above query is performed with indexing and the runtime is 82 milli seconds

- Filtering by Columns with High Cardinality
  Filtering on columns with high cardinality, such as rating or release year, can be slow when no indexes are applied, especially with large datasets. To address this, adding indexes either single column or composite based on common query patterns can significantly improve performance. For example, creating a single column index on release year speeds up filtering by year, while a composite index on rating and type optimizes queries that filter by both columns. These indexing strategies reduce query processing time and enhance the overall efficiency of data retrieval.

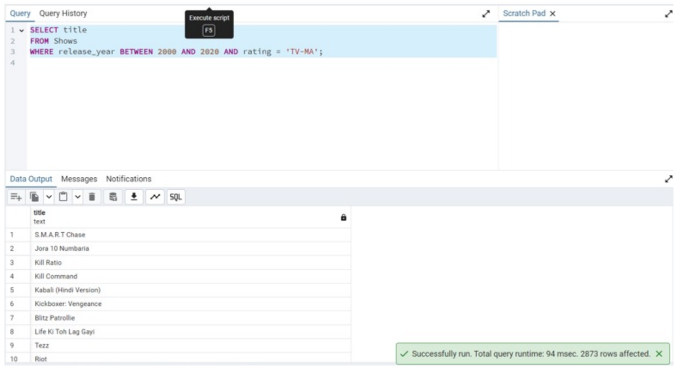


```
SELECT title
FROM Shows
WHERE release_year BETWEEN 2000 AND 2020 AND rating = 'TV-MA';
```

	title text
1	S.M.A.R.T Chase
2	Jora 10 Numbaria
3	Kill Ratio
4	Kill Command
5	Kabali (Hindi Version)
6	Kickboxer: Vengeance
7	Blitz Patrollie
8	Life Ki Toh Lag Gayi
9	Tezz
10	Riot

Successfully run. Total query runtime: 152 msec. 2873 rows affected.

Fig. 12. The above query is performed without indexing and the runtime is 152 milli seconds



```
SELECT title
FROM Shows
WHERE release_year BETWEEN 2000 AND 2020 AND rating = 'TV-MA';
```

	title text
1	S.M.A.R.T Chase
2	Jora 10 Numbaria
3	Kill Ratio
4	Kill Command
5	Kabali (Hindi Version)
6	Kickboxer: Vengeance
7	Blitz Patrollie
8	Life Ki Toh Lag Gayi
9	Tezz
10	Riot

Successfully run. Total query runtime: 94 msec. 2873 rows affected.

Fig. 13. The above query is performed with indexing and the runtime is 94 milli seconds

Yes, as shown in the above screenshots we used indexing concept to resolve the challenges faced while handling large datasets.

B. *Questions faced and solutions*

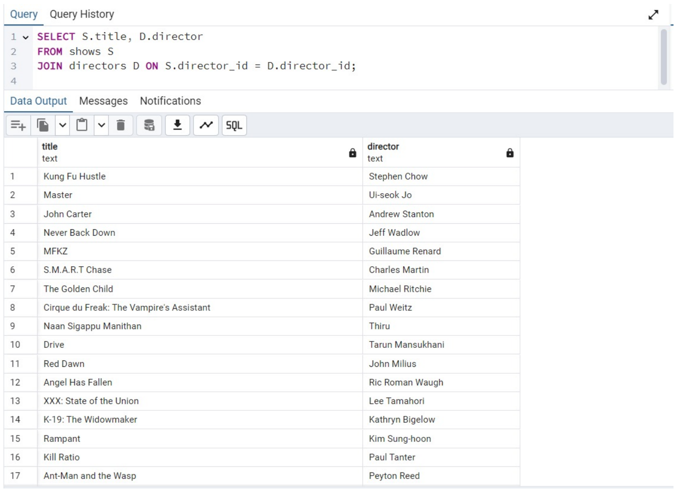
1) Improving join performance: Joins between tables like Shows, Actors, and other were slow due to the absence of indexes on foreign key columns. To address this, indexes were added on actor_id and show_id, enabling the database to perform efficient lookups instead of full table scans, significantly improving join performance.

2) Handling filtering by multiple columns: Filtering by rating and type in the Shows table was slow because the database could not effectively use multiple single column indexes. To optimize this, a composite index on rating and type was created, allowing the database to efficiently filter by both columns in a single operation.

3) Optimizing range queries: Range queries filtering by release_year were slow because they required scanning the entire Shows table. An index on release_year was created to improve query performance, and partial indexes were considered for targeting specific subsets, such as only TV Shows, for even greater optimization.

4) Challenges while using indexing: Indexes consume additional storage, especially with large dataset like the netflix dataset. To manage this, we indexed only the most frequently queried columns, avoiding unnecessary indexing of rarely used ones. Indexes slowed down operations like INSERT, UPDATE, and DELETE because they require updates during data changes. To avoid this indexing is temporarily disabled during bulk inserts, and indexes were rebuilt afterward for optimal performance.

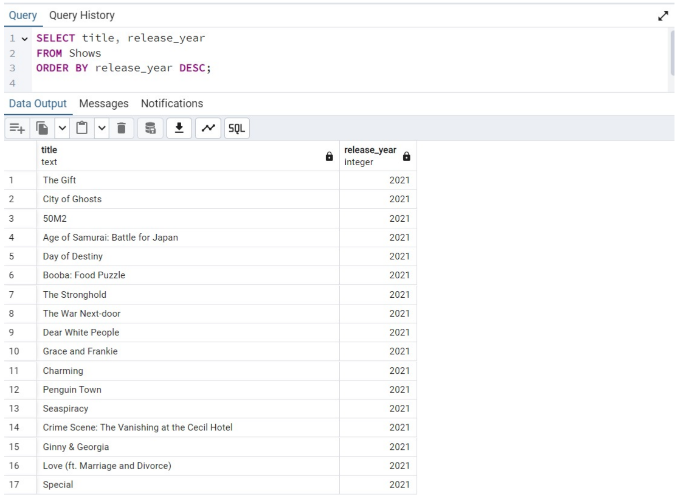
IX. SQL Queries and Results



```sql
SELECT S.title, D.director
FROM shows S
JOIN directors D ON S.director_id = D.director_id;
```

	title (text)	director (text)
1	Kung Fu Hustle	Stephen Chow
2	Master	Ui-seok Jo
3	John Carter	Andrew Stanton
4	Never Back Down	Jeff Wadlow
5	MFKZ	Guillaume Renard
6	S.M.A.R.T Chase	Charles Martin
7	The Golden Child	Michael Ritchie
8	Cirque du Freak: The Vampire's Assistant	Paul Weitz
9	Naan Sigappu Manithan	Thiru
10	Drive	Tarun Mansukhani
11	Red Dawn	John Milius
12	Angel Has Fallen	Ric Roman Waugh
13	XXX: State of the Union	Lee Tamahori
14	K-19: The Widowmaker	Kathryn Bigelow
15	Rampant	Kim Sung-hoon
16	Kill Ratio	Paul Tanter
17	Ant-Man and the Wasp	Peyton Reed

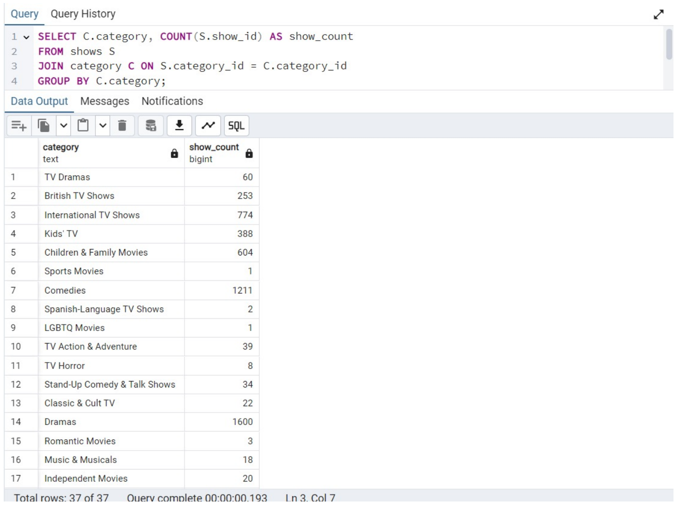
Fig. 14. The above query lists all shows along with their respective directors



```sql
SELECT title, release_year
FROM Shows
ORDER BY release_year DESC;
```

	title (text)	release_year (integer)
1	The Gift	2021
2	City of Ghosts	2021
3	50M2	2021
4	Age of Samurai: Battle for Japan	2021
5	Day of Destiny	2021
6	Booba: Food Puzzle	2021
7	The Stronghold	2021
8	The War Next-door	2021
9	Dear White People	2021
10	Grace and Frankie	2021
11	Charming	2021
12	Penguin Town	2021
13	Seaspiracy	2021
14	Crime Scene: The Vanishing at the Cecil Hotel	2021
15	Ginny & Georgia	2021
16	Love (ft. Marriage and Divorce)	2021
17	Special	2021

Fig. 15. The above query lists all shows ordered by release_year in descending order

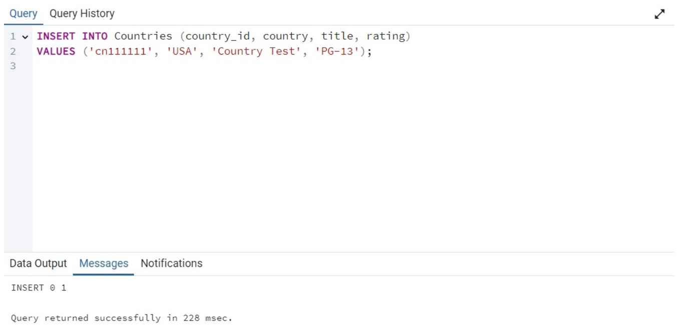


```sql
SELECT C.category, COUNT(S.show_id) AS show_count
FROM shows S
JOIN category C ON S.category_id = C.category_id
GROUP BY C.category;
```

	category (text)	show_count (bigint)
1	TV Dramas	60
2	British TV Shows	253
3	International TV Shows	774
4	Kids' TV	388
5	Children & Family Movies	604
6	Sports Movies	1
7	Comedies	1211
8	Spanish-Language TV Shows	2
9	LGBTQ Movies	1
10	TV Action & Adventure	39
11	TV Horror	8
12	Stand-Up Comedy & Talk Shows	34
13	Classic & Cult TV	22
14	Dramas	1600
15	Romantic Movies	3
16	Music & Musicals	18
17	Independent Movies	20

```
Total rows: 37 of 37    Query complete 00:00:00.193    Ln 3, Col 7
```

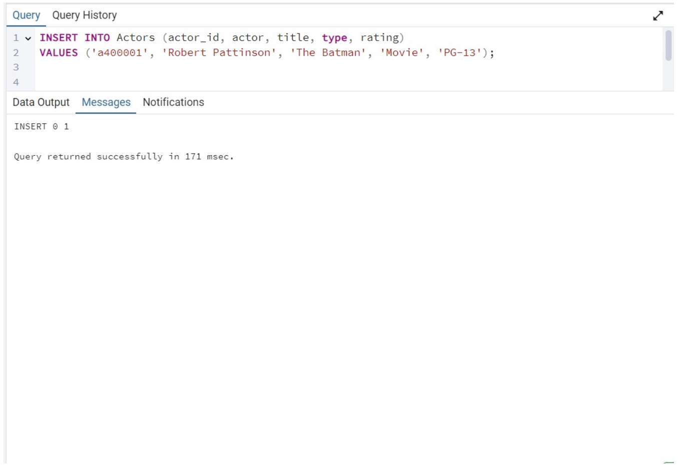
Fig. 16. The above query counts the number of shows for each category



```sql
INSERT INTO Countries (country_id, country, title, rating)
VALUES ('cn111111', 'USA', 'Country Test', 'PG-13');
```

```
INSERT 0 1

Query returned successfully in 228 msec.
```

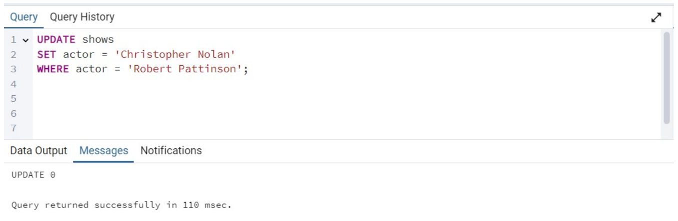
Fig. 17. The above query inserts the details of a new country into Countries table



```sql
INSERT INTO Actors (actor_id, actor, title, type, rating)
VALUES ('a400001', 'Robert Pattinson', 'The Batman', 'Movie', 'PG-13');
```

```
INSERT 0 1

Query returned successfully in 171 msec.
```

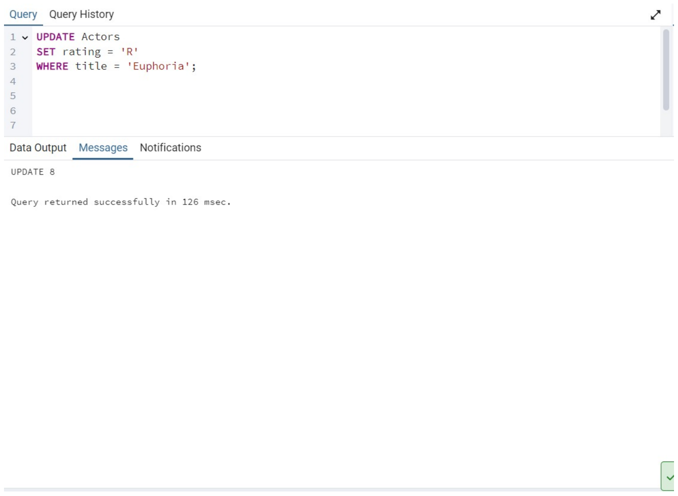
Fig. 18. The above query inserts details of a new actor into Actors table



```
UPDATE shows
SET actor = 'Christopher Nolan'
WHERE actor = 'Robert Pattinson';
```

```
UPDATE 0

Query returned successfully in 110 msec.
```

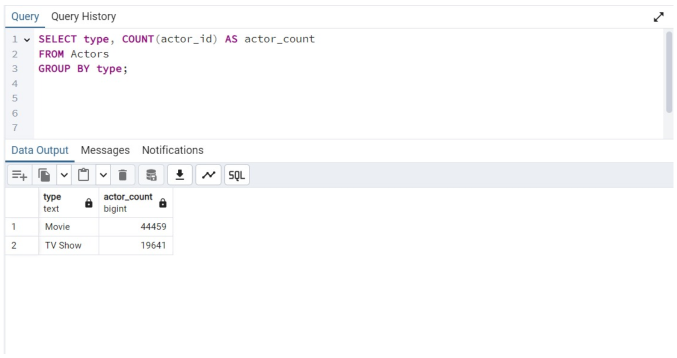
Fig. 19. The above query updates the details of an actor in Shows table



```
UPDATE Actors
SET rating = 'R'
WHERE title = 'Euphoria';
```

```
UPDATE 8

Query returned successfully in 126 msec.
```

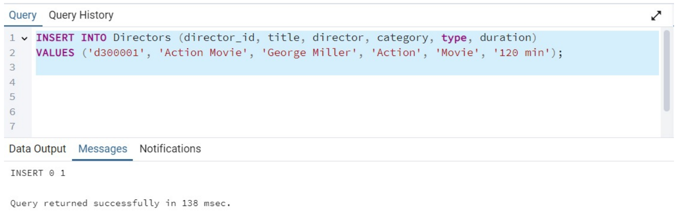
Fig. 20. The above query updates the rating of a title in Actors table



```
SELECT type, COUNT(actor_id) AS actor_count
FROM Actors
GROUP BY type;
```

	type text	actor_count bigint
1	Movie	44459
2	TV Show	19641

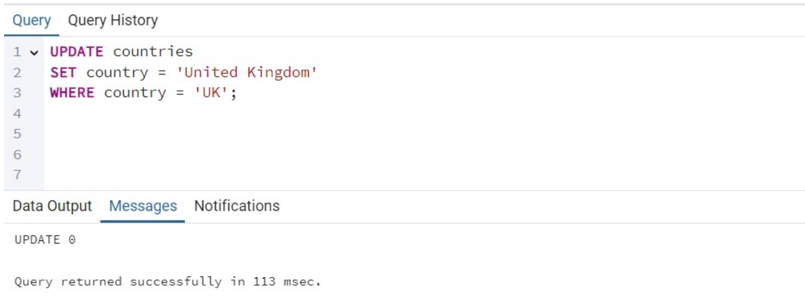
Fig. 21. The above query counts number of actors grouped by each type



```
INSERT INTO Directors (director_id, title, director, category, type, duration)
VALUES ('d300001', 'Action Movie', 'George Miller', 'Action', 'Movie', '120 min');
```

```
INSERT 0 1

Query returned successfully in 138 msec.
```

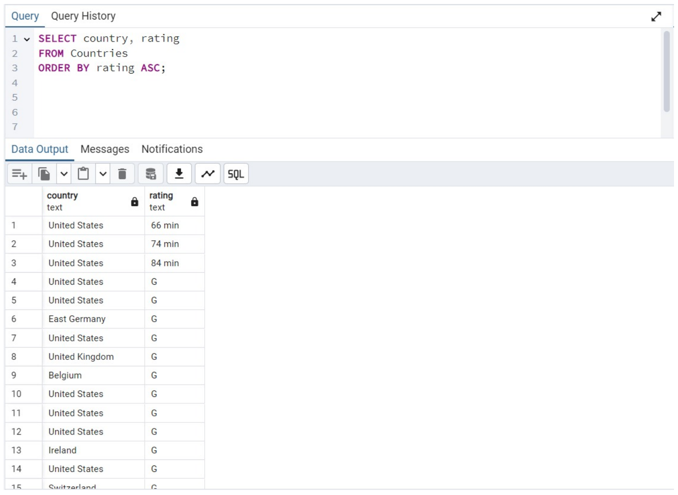
Fig. 22. The above query inserts the details of a new director into Directors table



```
UPDATE countries
SET country = 'United Kingdom'
WHERE country = 'UK';
```

```
UPDATE 0

Query returned successfully in 113 msec.
```

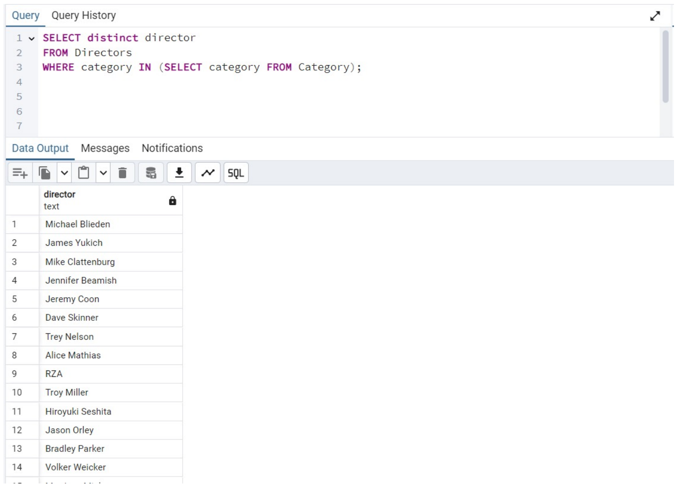
Fig. 23. The above query updates the country name



```
SELECT country, rating
FROM Countries
ORDER BY rating ASC;
```

	country text	rating text
1	United States	66 min
2	United States	74 min
3	United States	84 min
4	United States	G
5	United States	G
6	East Germany	G
7	United States	G
8	United Kingdom	G
9	Belgium	G
10	United States	G
11	United States	G
12	United States	G
13	Ireland	G
14	United States	G
15	Switzerland	G

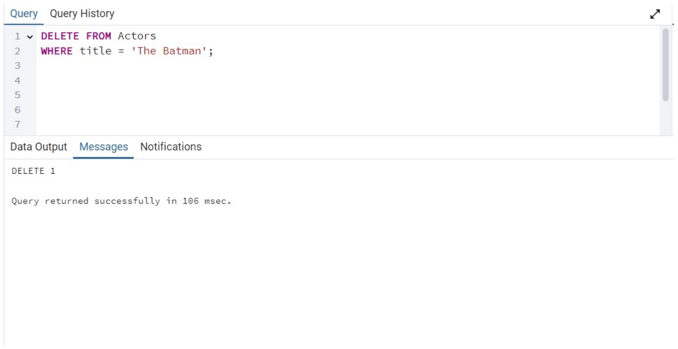
Fig. 24. The above query lists all countries ordered by their rating in ascending order



```sql
SELECT distinct director
FROM Directors
WHERE category IN (SELECT category FROM Category);
```

	director text
1	Michael Blieden
2	James Yukich
3	Mike Clattenburg
4	Jennifer Beamish
5	Jeremy Coon
6	Dave Skinner
7	Trey Nelson
8	Alice Mathias
9	RZA
10	Troy Miller
11	Hiroyuki Seshita
12	Jason Orley
13	Bradley Parker
14	Volker Weicker

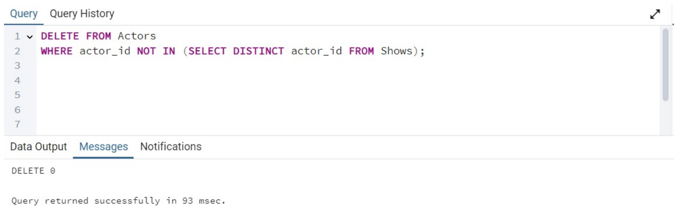
Fig. 25. The above query lists all directors whose category is referenced in the Category table



```sql
DELETE FROM Actors
WHERE title = 'The Batman';
```

```
DELETE 1

Query returned successfully in 106 msec.
```

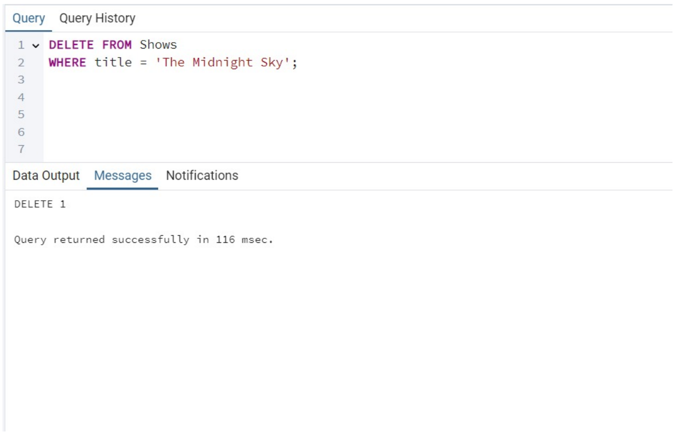
Fig. 26. The above query deletes the actors by title



```sql
DELETE FROM Actors
WHERE actor_id NOT IN (SELECT DISTINCT actor_id FROM Shows);
```

```
DELETE 0

Query returned successfully in 93 msec.
```

Fig. 27. The above query deletes the actors who are not referenced in any show



```sql
DELETE FROM Shows
WHERE title = 'The Midnight Sky';
```

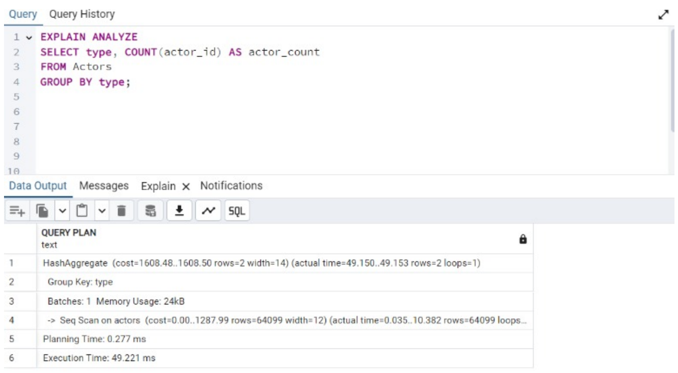
```
DELETE 1

Query returned successfully in 116 msec.
```

Fig. 28. The above query deletes the show by title

X. Query Execution Analysis

1) Aggregation:

- Aggregation before creating index
  - The below image shows the detailed execution plan of performing aggregation operation before creating index. The execution plan for aggregation before index creation shows two key operations: Hash Aggregate and Sequential Scan. The Hash Aggregate operation, costing 1608.48 to 1608.50, reflects the resources needed for grouping and counting, which can be resource-intensive. The Sequential Scan, with a cost of 0.00 to 1287.99, indicates a full scan of the Actors table with 64099 rows, as no indexes were available. The Planning Time was 0.277 ms, and the Execution Time was 49.221 ms, highlighting the performance impact of scanning a large table without indexes.



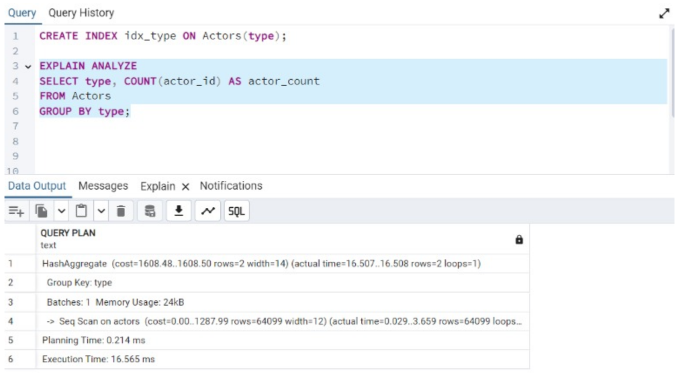
```sql
EXPLAIN ANALYZE
SELECT type, COUNT(actor_id) AS actor_count
FROM Actors
GROUP BY type;
```

```
   QUERY PLAN
   text
1  HashAggregate  (cost=1608.48..1608.50 rows=2 width=14) (actual time=49.150..49.153 rows=2 loops=1)
2    Group Key: type
3    Batches: 1  Memory Usage: 24kB
4    ->  Seq Scan on actors  (cost=0.00..1287.99 rows=64099 width=12) (actual time=0.035..10.382 rows=64099 loops...
5  Planning Time: 0.277 ms
6  Execution Time: 49.221 ms
```

Fig. 29. Aggregation before creating index

- Aggregation after creating index
  - The below image shows the detailed execution plan of performing aggregation operation after creating index. After index creation, the execution plan shows that the Hash Aggregate cost remains unchanged, as it is driven by the grouping logic, unaffected by indexing. The Sequential Scan still occurs, with a

cost between 0.00 and 1287.99, since aggregation processes all rows. However, the Execution Time improved significantly, dropping from 49.221 ms to 16.565 ms, demonstrating the performance boost from indexing. The Planning Time also slightly decreased from 0.277 ms to 0.214 ms, indicating a more efficient query planning process. Overall, indexing improved query performance even with a full table scan required for aggregation.



Query | Query History

```
CREATE INDEX idx_type ON Actors(type);

EXPLAIN ANALYZE
SELECT type, COUNT(actor_id) AS actor_count
FROM Actors
GROUP BY type;
```

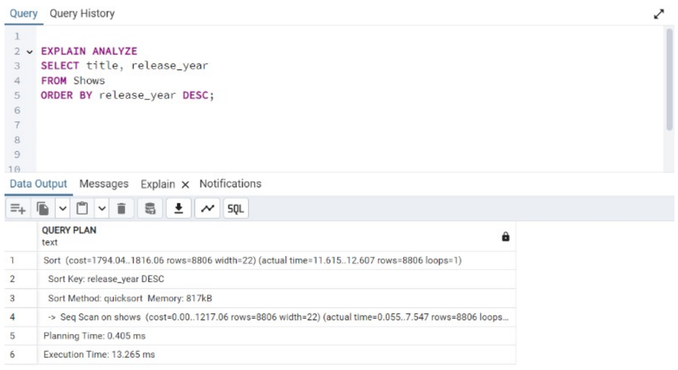
Data Output | Messages | Explain | Notifications

```
QUERY PLAN
text
1  HashAggregate (cost=1608.48..1608.50 rows=2 width=14) (actual time=16.507..16.508 rows=2 loops=1)
2    Group Key: type
3    Batches: 1  Memory Usage: 24kB
4    -> Seq Scan on actors (cost=0.00..1287.99 rows=64099 width=12) (actual time=0.029..3.659 rows=64099 loops...
5  Planning Time: 0.214 ms
6  Execution Time: 16.565 ms
```

Fig. 30. Aggregation after creating index

2) Sorting:

- Sorting before creating index
  - The below image shows the detailed execution plan of performing sorting operation before creating index. The execution plan for the Sort Operation without an index shows a Startup Cost of 1794.04 and a Total Cost of 1816.06, reflecting the resources required for sorting. The sort is performed on the release_year column in descending order using the quick sort algorithm, with a memory usage of 817 kB. The Actual Time for sorting is 11.615 ms to start and 12.607 ms to complete. The Sequential Scan has a Startup Cost of 0.00 and a Total Cost of 1217.06, indicating a full scan of all 8806 rows in the Shows table. The scan takes 0.055 ms to begin and 7.547 ms to complete. Planning Time of 0.405 ms and a total Execution Time of 13.265 ms for the query demonstrates the performance impact of not having indexes, resulting in slower sorting and scanning operations.



Query | Query History

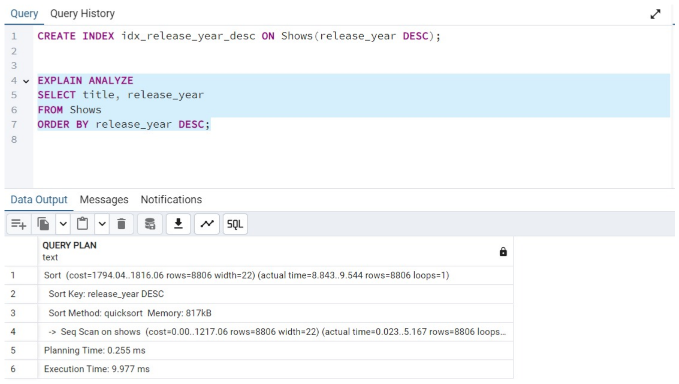
```
EXPLAIN ANALYZE
SELECT title, release_year
FROM Shows
ORDER BY release_year DESC;
```

Data Output | Messages | Explain | Notifications

```
QUERY PLAN
text
1  Sort (cost=1794.04..1816.06 rows=8806 width=22) (actual time=11.615..12.607 rows=8806 loops=1)
2    Sort Key: release_year DESC
3    Sort Method: quicksort  Memory: 817kB
4    -> Seq Scan on shows (cost=0.00..1217.06 rows=8806 width=22) (actual time=0.055..7.547 rows=8806 loops...
5  Planning Time: 0.405 ms
6  Execution Time: 13.265 ms
```

Fig. 31. Sorting before creating index

- Sorting after creating index
  - The below image shows the detailed execution plan of performing sorting operation after creating index. The execution plan after indexing shows a Startup Cost of 1794.04 and a Total Cost of 1816.06 for the Sort Operation. Sorting is performed on the release_year column in descending order using the quick sort algorithm, with 817 kB of memory used. The sorting time is 8.843 ms to start and 9.544 ms to complete. The Sequential Scan has a Total Cost of 1217.06, processing all 8806 rows in the Shows table in 5.167 ms. The Planning Time is 0.255 ms, and the Execution Time is 9.977 ms. Indexing significantly improves performance, reducing the overall query execution time and optimizing sorting and scanning.



Query | Query History

```
CREATE INDEX idx_release_year_desc ON Shows(release_year DESC);


EXPLAIN ANALYZE
SELECT title, release_year
FROM Shows
ORDER BY release_year DESC;
```

Data Output | Messages | Notifications

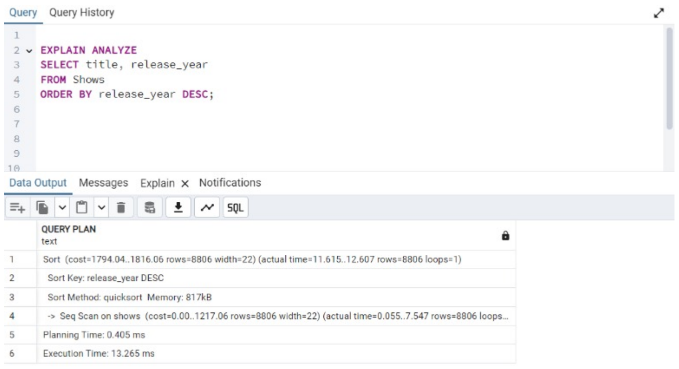
```
QUERY PLAN
text
1  Sort (cost=1794.04..1816.06 rows=8806 width=22) (actual time=8.843..9.544 rows=8806 loops=1)
2    Sort Key: release_year DESC
3    Sort Method: quicksort  Memory: 817kB
4    -> Seq Scan on shows (cost=0.00..1217.06 rows=8806 width=22) (actual time=0.023..5.167 rows=8806 loops...
5  Planning Time: 0.255 ms
6  Execution Time: 9.977 ms
```

Fig. 32. Sorting after creating index

3) Sub-query optimization:

- Sub-query optimization before creating index
  - The below image shows the detailed execution plan of performing Sub-query optimization before creating index. Before index creation, the query plan analysis reveals a Nested Loop operation with a Startup Cost of 0.29 and a Total Cost of 222.49. The Actual Time for the nested loop operation was

3.606 to 3.607 ms, as it iterated over rows to join the Shows and Countries tables. The Sequential Scan on the Countries table, costing 0.00 to 214.18, took 3.547 to 3.549 ms to scan all rows and filter by country. The Total Execution Time was 3.666 ms, indicating that performance was limited due to the lack of indexing.

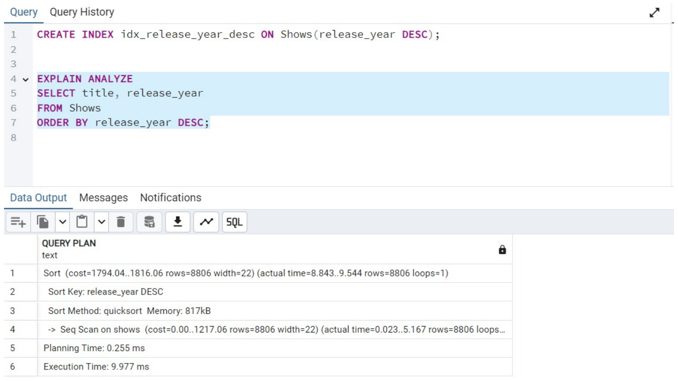


```
EXPLAIN ANALYZE
SELECT title, release_year
FROM Shows
ORDER BY release_year DESC;
```

```
QUERY PLAN
text
1  Sort  (cost=1794.04..1816.06 rows=8806 width=22) (actual time=11.615..12.607 rows=8806 loops=1)
2    Sort Key: release_year DESC
3    Sort Method: quicksort  Memory: 817kB
4    -> Seq Scan on shows  (cost=0.00..1217.06 rows=8806 width=22) (actual time=0.055..7.547 rows=8806 loops...
5  Planning Time: 0.405 ms
6  Execution Time: 13.265 ms
```

Fig. 33. Sub-query optimization before creating index

- Sub-query optimization after creating index
  - The below image shows the detailed execution plan of performing Sub-query optimization after creating index. After index creation, the query plan analysis shows a Nested Loop operation with a Startup Cost of 0.29 and a Total Cost of 222.49, with no change in the cost. The actual time for the nested loop improved to 2.026 ms, reflecting faster lookups in the Shows table due to the newly created index. The Sequential Scan on the Countries table still required scanning all rows, with no change in the actual time 1.990 to 1.991 ms. The introduction of an Index Scan on the Shows table, utilizing the idx_country_id index, dramatically reduced the lookup time to 0.030 to 0.031 ms, significantly improving query performance. The Total Execution Time was reduced to 2.071 ms, showing a clear enhancement in query performance after indexing.



```
CREATE INDEX idx_release_year_desc ON Shows(release_year DESC);


EXPLAIN ANALYZE
SELECT title, release_year
FROM Shows
ORDER BY release_year DESC;
```

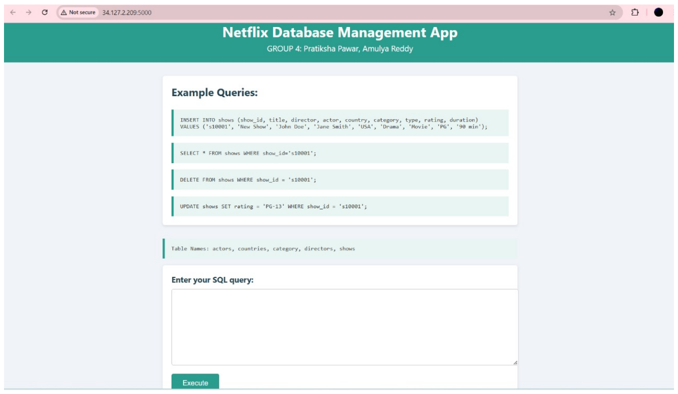
```
QUERY PLAN
text
1  Sort  (cost=1794.04..1816.06 rows=8806 width=22) (actual time=8.843..9.544 rows=8806 loops=1)
2    Sort Key: release_year DESC
3    Sort Method: quicksort  Memory: 817kB
4    -> Seq Scan on shows  (cost=0.00..1217.06 rows=8806 width=22) (actual time=0.023..5.167 rows=8806 loops...
5  Planning Time: 0.255 ms
6  Execution Time: 9.977 ms
```

Fig. 34. Sub-query optimization after creating index

XI. BONUS

http://34.127.2.209:5000/

The above is the link to the website linking with our database to display or visualize query and query results for Netflix Dataset.



Netflix Database Management App

GROUP 4: Pratiksha Pawar, Amulya Reddy

Example Queries:

```
INSERT INTO shows (show_id, title, director, actor, country, category, type, rating, duration)
VALUES ('s10001', 'New Show', 'John Doe', 'Jane Smith', 'USA', 'Drama', 'Movie', 'PG', '90 min');

SELECT * FROM shows WHERE show_id='s10001';

DELETE FROM shows WHERE show_id = 's10001';

UPDATE shows SET rating = 'PG-13' WHERE show_id = 's10001';
```

Table Names: actors, countries, category, directors, shows

Enter your SQL query:

Fig. 35. Screenshot-1 of the website

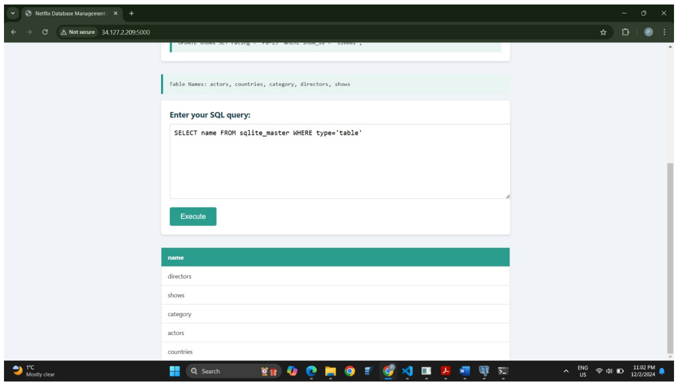


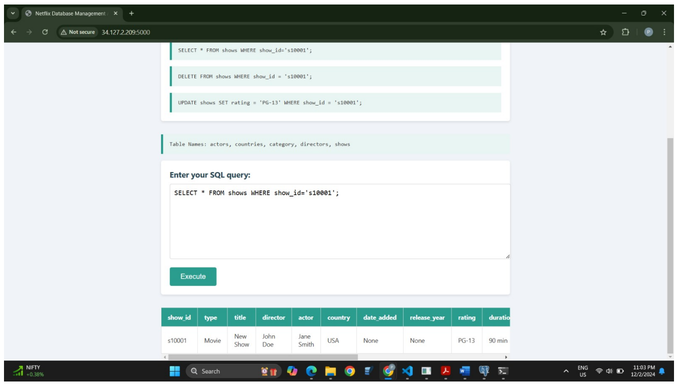
Table Names: actors, countries, category, directors, shows

Enter your SQL query:

```
SELECT name FROM sqlite_master WHERE type='table'
```

name
directors
shows
category
actors
countries

Fig. 36. Screenshot-2 of the website



```
SELECT * FROM shows WHERE show_id='s10001';
```

```
DELETE FROM shows WHERE show_id = 's10001';
```

```
UPDATE shows SET rating = 'PG-13' WHERE show_id = 's10001';
```

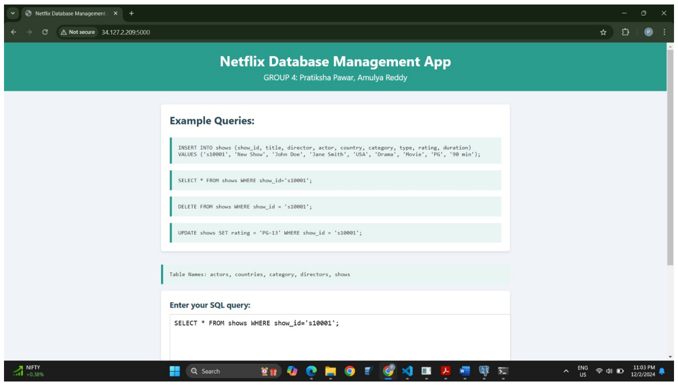
```
Table Names: actors, countries, category, directors, shows
```

Enter your SQL query:

```
SELECT * FROM shows WHERE show_id='s10001';
```

show_id	type	title	director	actor	country	date_added	release_year	rating	duratio
s10001	Movie	New Show	John Doe	Jane Smith	USA	None	None	PG-13	90 min

Fig. 37. Screenshot-3 of the website



Netflix Database Management App

GROUP 4: Pratiksha Pawar, Amulya Reddy

Example Queries:

```
INSERT INTO shows (show_id, title, director, actor, country, category, type, rating, duration)
VALUES ('s10001', 'New Show', 'John Doe', 'Jane Smith', 'USA', 'Drama', 'Movie', 'PG', '90 min');
```

```
SELECT * FROM shows WHERE show_id='s10001';
```

```
DELETE FROM shows WHERE show_id = 's10001';
```

```
UPDATE shows SET rating = 'PG-13' WHERE show_id = 's10001';
```

```
Table Names: actors, countries, category, directors, shows
```

Enter your SQL query:

```
SELECT * FROM shows WHERE show_id='s10001';
```

Fig. 38. Screenshot-4 of the website

The above four images are the screenshots of the running website that links with our netflix database displaying queries and query results

XII. References

.

Netflix Dataset on Kaggle

Milestone-1 files(UB Box)