Cinematic Insight Odyssey

Team : Insight Illuminati

Pranav Polavarapu (pranavpo) Sai Keerthi Chelluri (schellur) Nikhil Raj Yammani (nyammani)

I. Introduction & Milestone-1 Overview

A. Project Aim

The " Cinematic Insight Odyssey " aims to develop a comprehensive movie database to enhance the movie-watching experience by providing detailed insights into various cinematic elements. This robust platform enables industry professionals to access extensive data about movies, facilitating strategic decision-making in film production and marketing.

B. Problem Statement

The dynamic nature of movie data requires a specialized database, superior to simpler tools like spreadsheets. Our database efficiently handles a vast array of data elements, from basic movie details to intricate crew collaborations, sourced from IMDB. It offers a structured and scalable solution for rapid data retrieval and high integrity.

C. Business Insights

Our project delivers crucial data-driven insights for:

For Marketing Teams and Content Developers:

- Tailoring marketing strategies and content to specific audience preferences.
- Enhancing engagement through targeted campaigns and personalized content.

For Production Houses:

- Providing insights on genre popularity and directorial success.
- Informing better production decisions based on data-driven analytics.

For Streaming Platforms:

- Supporting recommendation algorithms to improve content discoverability.
- Personalizing viewer recommendations to enhance user experience.

D. Initial Schema and Data Source

1) Primary Data Source: The project utilizes the IMDB Non-commercial Datasets, which consist of seven gzipped TSV files updated daily. These files cover extensive movie-related data, including genres, ratings, crew and cast information, and alternative titles. The specific files are:

`name.basics.tsv.gz | title.akas.tsv.gz | title.basics.tsv.gz | title.crew.tsv.gz | title.episode.tsv.gz | title.principals.tsv.gz | title.ratings.tsv.gz`

This rich dataset requires a well-structured relational database to facilitate complex queries for insightful analysis.

2) Schema and E/R Diagram: Our proposed database schema organizes the diverse IMDB data into 11 relational tables designed to minimize redundancy and optimize query performance. These tables include "Movies", "Genre", "People", "Occupations", "MovieGenres", "CastAndCrew", "AlternateTitles", "Title-Crew", "Episodes", "FilmRoles", and "Rating". This relational structure supports many-to-many relationships between movies and genres, as well as movies and various personnel, facilitated through linking tables such as "moviegenres" and "castandcrew". The initial E/R diagram provided in the project outlines these relationships and serves as a foundational blueprint for our database design.

- **Movies**: Holds essential details like title, release years, and runtime, central to linking genres, people, and ratings for trend analysis.
- **Genre**: Classifies movies into genres, aiding in popularity and performance analyses.
- **MovieGenres**: Connects movies to their genres, useful for demographic and market studies.
- **People**: Tracks industry figures (actors, directors) and their career data, enabling demographic and impact studies.
- **Occupations**: Enumerates job roles in film, supporting employment trend analysis.
- **CastAndCrew**: Maps individuals to their roles in movies, key for collaboration and impact analysis.
- **AlternateTitles**: Records movie titles across regions, highlighting global reach and regional preferences.
- **TitleCrew**: Associates movies with their creative heads, crucial for studying the influence on film success.
- **Episodes**: Details on TV series episodes, useful for viewership and series popularity analysis.
- **FilmRoles**: Links actors to their movie characters, useful for studying actor versatility and typecasting.
- **Rating**: Contains movie ratings and votes, essential for public reception and popularity trends analysis.

E. Data Loading and Transformation

1) Transformation Process: Given the complexity and format of the raw IMDB data, significant transformation is necessary to make the data compatible with our relational schema. We used Python scripts utilizing libraries such as pandas, csv, and os to handle these transformations. These scripts restructured the raw TSV data into a format suitable for our database, addressing challenges like multiple feature combinations spread across only a few files and ensuring correct feature mapping for our proposed schema.

2) Loading Process: After transformation, the data was loaded into the database using PostgreSQL \copy commands. This process involved creating new TSV files for each of the 11 tables, ensuring that each table's schema definitions matched the newly transformed data files. This ensured that the database is populated with accurate and query-ready data.

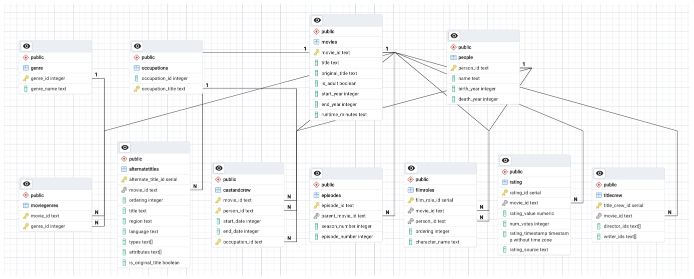
3) Challenges and Solutions: One major challenge was ensuring data integrity and consistency during the transformation process, especially when splitting and recombining data from the source files to fit our relational model. The solution involved rigorous testing and iterative refinement of our Python scripts to ensure they accurately mapped the data to the correct tables without loss of information.

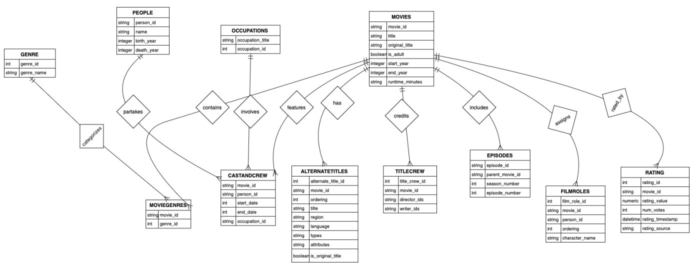
II. Database Schema Refinement

Refer to **TABLE 1**

Conclusion - Normalization All tables in the schema are in BCNF, based on the functional dependencies present. No tables require decomposition.

III. E/R Diagram





A. Transformations from Milestone 1

1) Normalization: Followed Database Design Rules to ensure that each table adhered to BCNF, thus reducing redundancy and preventing update anomalies.

2) Data Type Refinement: We noticed that certain fields, such as `is_adult`, `start_year`, and `end_year`, were initially text types. To streamline our database and enforce data integrity, we updated these fields to more appropriate data types—boolean and integer respectively—enabling more efficient storage and querying.

3) Linking Tables Creation: To properly represent many-to-many relationships—such as those between movies and genres, and movies and people—we introduced linking tables like `moviegenres` and `castandcrew`. This change allows our database to more accurately and flexibly represent complex associations without redundancy.

4) Primary and Foreign Keys: We solidified the relationships between tables with relevant primary and foreign key constraints. This ensured that all references from one table to another are valid and consistent.

B. Constraints and Checks Implemented

1) Primary Key Constraints:

- **movies:** `movie_id` serves as the primary key to uniquely identify each movie.
- **genre:** `genre_id` is the unique identifier for each genre.
- **people:** `person_id` uniquely identifies each person involved in the movies.
- Other tables have their own primary keys ensuring uniqueness within their records.

2) Foreign Key Constraints:

- **moviegenres:**
  - `movie_id` references `movies(movie_id)` ensuring linked records exist.
  - `genre_id` references `genre(genre_id)` maintaining valid genre associations.

3) ON DELETE SET NULL::

- In `moviegenres`, if a movie or genre gets deleted, its ID turns to NULL in this table.
- Prevents broken links by not having references to deleted movies or genres.

4) ON UPDATE CASCADE::

- If a `movie_id` is changed in `movies`, it updates automatically in `moviegenres`.
- Keeps all relationships up-to-date and accurate even after changes.

5) Data Type Constraints:

- **Arrays:** The `types` column in `alternatetitles` and `director_ids`, `writer_ids` in `titlecrew` use array data types to store multiple values.
- **Numerics:** `rating_value` in the `rating` table uses numeric data type for precise representation of ratings.

6) Unique Constraints:: Implicit through primary keys on each table.

TABLE I
DATABASE TABLES WITH FUNCTIONAL DEPENDENCIES AND BCNF CHECK

Table	Primary Key	Functional Dependencies	BCNF Check
movies	movie_id	movie_id → title, original_title, is_adult, start_year, end_year, runtime_minutes	The primary key determines all attributes. This table is in BCNF.
genre	genre_id	genre_id → genre_name	The primary key determines all attributes. This table is in BCNF.
moviegenres	(movie_id, genre_id)	(movie_id, genre_id) → None	The primary key determines all attributes. This table is in BCNF.
people	person_id	person_id → name, birth_year, death_year	The primary key determines all attributes. This table is in BCNF.
occupations	occupation_title	occupation_title → occupation_id	The primary key determines all attributes. This table is in BCNF.
castandcrew	(movie_id, person_id, occupation_id)	(movie_id, person_id, occupation_id) → start_date, end_date	The primary key determines all attributes. This table is in BCNF.
alternatetitles	alternate_title_id	alternate_title_id → movie_id, ordering, title, region, language, types, attributes, is_original_title	The primary key determines all attributes. This table is in BCNF.
titlecrew	title_crew_id	title_crew_id → movie_id, director_ids, writer_ids	The primary key determines all attributes. This table is in BCNF.
episodes	episode_id	episode_id → parent_movie_id, season_number, episode_number	The primary key determines all attributes. This table is in BCNF.
filmroles	film_role_id	film_role_id → movie_id, person_id, ordering, character_name	The primary key determines all attributes. This table is in BCNF.
rating	rating_id	rating_id → movie_id, rating_value, num_votes, rating_timestamp, rating_source	The primary key determines all attributes. This table is in BCNF.

IV. HANDLING LARGE DATASETS

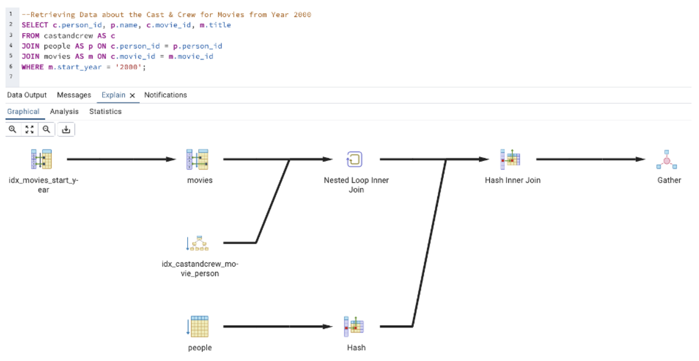
Query Slowdown: Experienced lag in retrieving cast and crew details by year.

Indexing: Applied a bitmap index on `movies.start_year` and a composite index on `castandcrew.movie_id` and `person_id`.

Analysis: Reviewed query plans with `EXPLAIN ANALYZE`; saw effective index use and parallel processing.

Outcome: Improved query speed with multi-core support.

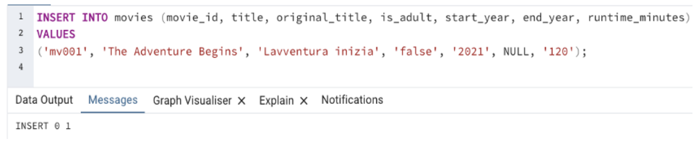
Based on this, we observed that Indexing is quite helpful in speeding up performance. Hence, we Indexed Frequently Queried Columns, Used Composite Indexes, and Optimized JOIN Operations.



```sql
--Retrieving Data about the Cast & Crew for Movies from Year 2000
SELECT c.person_id, p.name, c.movie_id, m.title
FROM castandcrew AS c
JOIN people AS p ON c.person_id = p.person_id
JOIN movies AS m ON c.movie_id = m.movie_id
WHERE m.start_year = '2000';
```

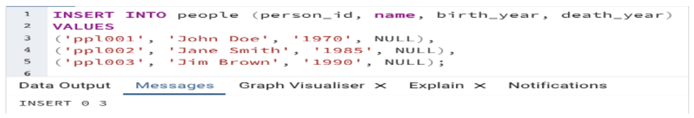
V. SQL QUERIES TESTING

A. Insert Queries (2 Queries)



```sql
INSERT INTO movies (movie_id, title, original_title, is_adult, start_year, end_year, runtime_minutes)
VALUES
('mv001', 'The Adventure Begins', 'Lavventura inizia', 'false', '2021', NULL, '120');
```

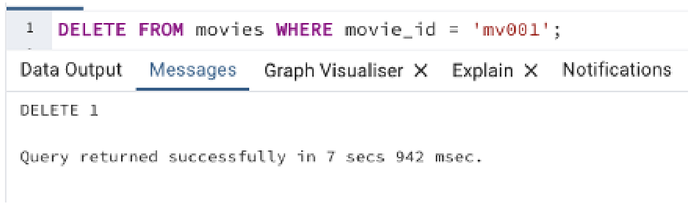
```
INSERT 0 1
```



```sql
INSERT INTO people (person_id, name, birth_year, death_year)
VALUES
('ppl001', 'John Doe', '1970', NULL),
('ppl002', 'Jane Smith', '1985', NULL),
('ppl003', 'Jim Brown', '1990', NULL);
```

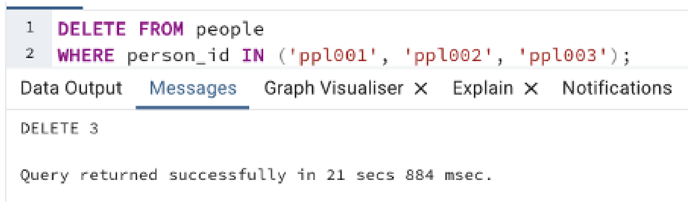
```
INSERT 0 3
```

B. Delete Queries (2 Queries)



```sql
DELETE FROM movies WHERE movie_id = 'mv001';
```

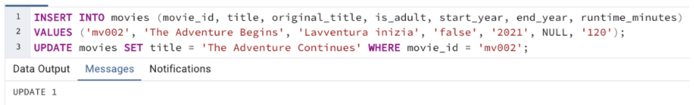
```
DELETE 1

Query returned successfully in 7 secs 942 msec.
```



```sql
DELETE FROM people
WHERE person_id IN ('ppl001', 'ppl002', 'ppl003');
```

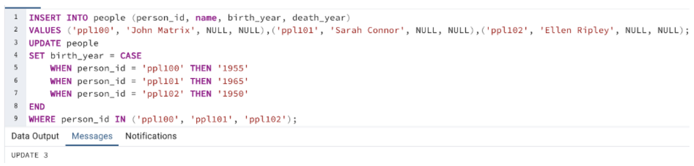
```
DELETE 3

Query returned successfully in 21 secs 884 msec.
```

C. Update Queries (2 Queries)



```sql
INSERT INTO movies (movie_id, title, original_title, is_adult, start_year, end_year, runtime_minutes)
VALUES ('mv002', 'The Adventure Begins', 'Lavventura inizia', 'false', '2021', NULL, '120');
UPDATE movies SET title = 'The Adventure Continues' WHERE movie_id = 'mv002';
```

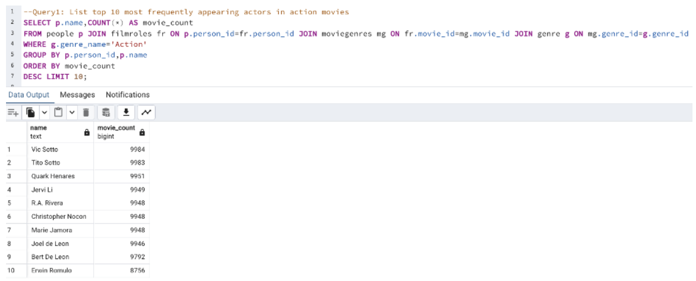
```
UPDATE 1
```



```sql
INSERT INTO people (person_id, name, birth_year, death_year)
VALUES ('ppl100', 'John Matrix', NULL, NULL),('ppl101', 'Sarah Connor', NULL, NULL),('ppl102', 'Ellen Ripley', NULL, NULL);
UPDATE people
SET birth_year = CASE
    WHEN person_id = 'ppl100' THEN '1955'
    WHEN person_id = 'ppl101' THEN '1965'
    WHEN person_id = 'ppl102' THEN '1950'
END
WHERE person_id IN ('ppl100', 'ppl101', 'ppl102');
```

```
UPDATE 3
```

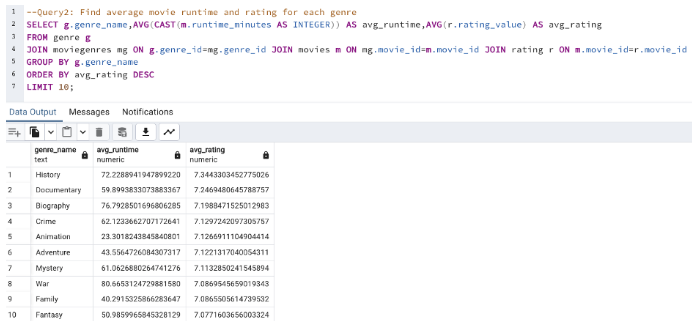
D. *SELECT Queries (4 Queries)*



```sql
--Query1: List top 10 most frequently appearing actors in action movies
SELECT p.name,COUNT(*) AS movie_count
FROM people p JOIN filmroles fr ON p.person_id=fr.person_id JOIN moviegenres mg ON fr.movie_id=mg.movie_id JOIN genre g ON mg.genre_id=g.genre_id
WHERE g.genre_name='Action'
GROUP BY p.person_id,p.name
ORDER BY movie_count
DESC LIMIT 10;
```

Data Output Messages Notifications

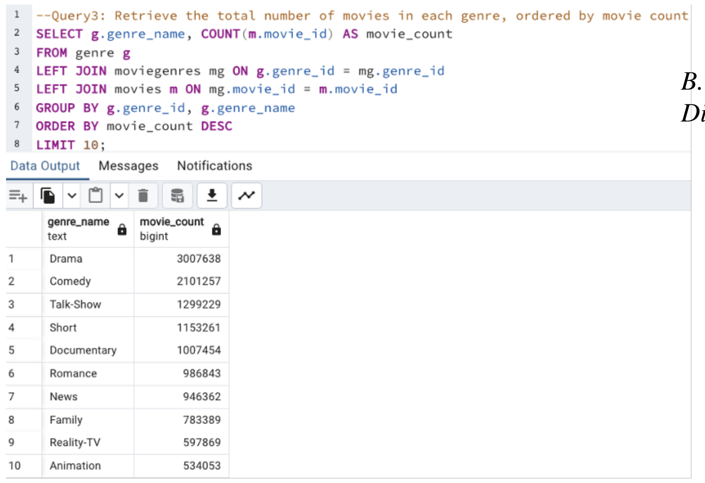
	name text	movie_count bigint
1	Vic Sotto	9984
2	Tito Sotto	9983
3	Quark Henares	9951
4	Jervi Li	9949
5	R.A. Rivera	9948
6	Christopher Nocon	9948
7	Marie Jamora	9948
8	Joel de Leon	9946
9	Bert De Leon	9792
10	Erwin Romulo	8756



```sql
--Query2: Find average movie runtime and rating for each genre
SELECT g.genre_name,AVG(CAST(m.runtime_minutes AS INTEGER)) AS avg_runtime,AVG(r.rating_value) AS avg_rating
FROM genre g
JOIN moviegenres mg ON g.genre_id=mg.genre_id JOIN movies m ON mg.movie_id=m.movie_id JOIN rating r ON m.movie_id=r.movie_id
GROUP BY g.genre_name
ORDER BY avg_rating DESC
LIMIT 10;
```

Data Output Messages Notifications

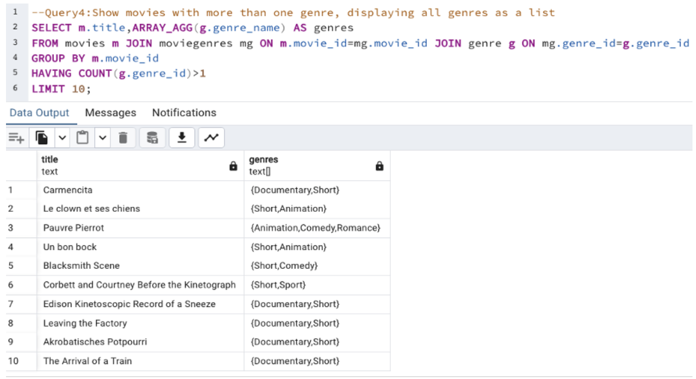
	genre_name text	avg_runtime numeric	avg_rating numeric
1	History	72.2288941947899220	7.3443303452775026
2	Documentary	59.8993833073883367	7.2469480645788757
3	Biography	76.7928501696806285	7.1988471525012983
4	Crime	62.1233662707172641	7.1297242097305757
5	Animation	23.3018243845840801	7.1266911104904414
6	Adventure	43.5564726084307317	7.1221317040054311
7	Mystery	61.0626880264741276	7.1132850241545894
8	War	80.6653124729881580	7.0869545659019343
9	Family	40.2915325866283647	7.0865505614739532
10	Fantasy	50.9859965845328129	7.0771603656003324



```sql
--Query3: Retrieve the total number of movies in each genre, ordered by movie count
SELECT g.genre_name, COUNT(m.movie_id) AS movie_count
FROM genre g
LEFT JOIN moviegenres mg ON g.genre_id = mg.genre_id
LEFT JOIN movies m ON mg.movie_id = m.movie_id
GROUP BY g.genre_id, g.genre_name
ORDER BY movie_count DESC
LIMIT 10;
```

Data Output Messages Notifications

	genre_name text	movie_count bigint
1	Drama	3007638
2	Comedy	2101257
3	Talk-Show	1299229
4	Short	1153261
5	Documentary	1007454
6	Romance	986843
7	News	946362
8	Family	783389
9	Reality-TV	597869
10	Animation	534053



```sql
--Query4:Show movies with more than one genre, displaying all genres as a list
SELECT m.title,ARRAY_AGG(g.genre_name) AS genres
FROM movies m JOIN moviegenres mg ON m.movie_id=mg.movie_id JOIN genre g ON mg.genre_id=g.genre_id
GROUP BY m.movie_id
HAVING COUNT(g.genre_id)>1
LIMIT 10;
```

Data Output Messages Notifications

	title text	genres text[]
1	Carmencita	{Documentary,Short}
2	Le clown et ses chiens	{Short,Animation}
3	Pauvre Pierrot	{Animation,Comedy,Romance}
4	Un bon bock	{Short,Animation}
5	Blacksmith Scene	{Short,Comedy}
6	Corbett and Courtney Before the Kinetograph	{Short,Sport}
7	Edison Kinetoscopic Record of a Sneeze	{Documentary,Short}
8	Leaving the Factory	{Documentary,Short}
9	Akrobatisches Potpourri	{Documentary,Short}
10	The Arrival of a Train	{Documentary,Short}

VI. Query Execution Analysis - EXPLAIN ANALYSIS

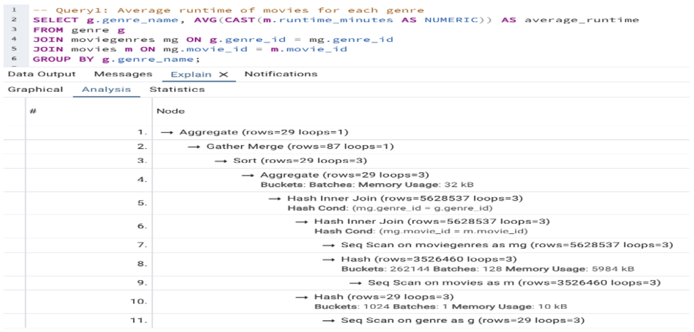
A. *Query 1: Average Runtime of Movies for Each Genre*

1) *Problematic Aspects:*

- High number of loops in Seq Scans and Joins.
- Aggregate and Gather Merge operations are performed on a large dataset.

2) *Performance Improvements:*

- Use indexes on `genre_id` columns in both `movies` and `genre` tables.
- Consider materialized views if the average runtimes are frequently queried and don't change often.
- Review and optimize `CAST` operation to ensure it's not adding unnecessary overhead.



```sql
-- Query1: Average runtime of movies for each genre
SELECT g.genre_name, AVG(CAST(m.runtime_minutes AS NUMERIC)) AS average_runtime
FROM genre g
JOIN moviegenres mg ON g.genre_id = mg.genre_id
JOIN movies m ON mg.movie_id = m.movie_id
GROUP BY g.genre_name;
```

Data Output Messages Explain Notifications
Graphical Analysis Statistics

```
#    Node
1.   → Aggregate (rows=29 loops=1)
2.       → Gather Merge (rows=87 loops=1)
3.           → Sort (rows=29 loops=3)
4.               → Aggregate (rows=29 loops=3)
                   Buckets: Batches: Memory Usage: 32 kB
5.                   → Hash Inner Join (rows=5628537 loops=3)
                       Hash Cond: (mg.genre_id = g.genre_id)
6.                       → Hash Inner Join (rows=5628537 loops=3)
                           Hash Cond: (mg.movie_id = m.movie_id)
7.                           → Seq Scan on moviegenres as mg (rows=5628537 loops=3)
8.                           → Hash (rows=3526460 loops=3)
                               Buckets: 262144 Batches: 128 Memory Usage: 5984 kB
9.                               → Seq Scan on movies as m (rows=3526460 loops=3)
10.                      → Hash (rows=29 loops=3)
                           Buckets: 1024 Batches: 1 Memory Usage: 10 kB
11.                          → Seq Scan on genre as g (rows=29 loops=3)
```

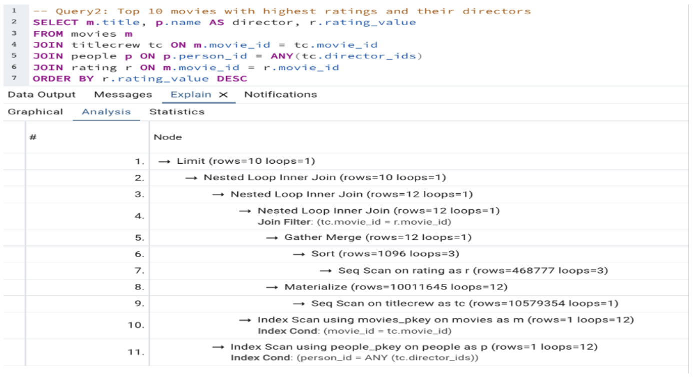
B. *Query 2: Top 10 Movies with Highest Ratings and Their Directors*

1) *Problematic Aspects:*

- Nested Loop Inner Joins suggest absence of indexes.
- Large table sequential scans indicate potential missing indexes.
- Sort operation for ordering which could be costly on a large dataset.

2) *Performance Improvements:*

- Index `movie_id` and `director_ids` to avoid Seq Scans.
- Implement indexes on `rating_value` and `movie_id` if the query is common.
- Use `LIMIT` with `JOINs` carefully, ensuring that `LIMIT` is applied after joins to avoid unnecessary processing.



```sql
-- Query2: Top 10 movies with highest ratings and their directors
SELECT m.title, p.name AS director, r.rating_value
FROM movies m
JOIN titlecrew tc ON m.movie_id = tc.movie_id
JOIN people p ON p.person_id = ANY(tc.director_ids)
JOIN rating r ON m.movie_id = r.movie_id
ORDER BY r.rating_value DESC
```

Data Output Messages Explain Notifications
Graphical Analysis Statistics

```
#    Node
1.   → Limit (rows=10 loops=1)
2.       → Nested Loop Inner Join (rows=10 loops=1)
3.           → Nested Loop Inner Join (rows=12 loops=1)
4.               → Nested Loop Inner Join (rows=12 loops=1)
                   Join Filter: (tc.movie_id = r.movie_id)
5.                   → Gather Merge (rows=12 loops=1)
6.                       → Sort (rows=1096 loops=3)
7.                           → Seq Scan on rating as r (rows=468777 loops=3)
8.                   → Materialize (rows=10011645 loops=12)
9.                       → Seq Scan on titlecrew as tc (rows=10579354 loops=1)
10.              → Index Scan using movies_pkey on movies as m (rows=1 loops=12)
                   Index Cond: (movie_id = tc.movie_id)
11.          → Index Scan using people_pkey on people as p (rows=1 loops=12)
               Index Cond: (person_id = ANY (tc.director_ids))
```

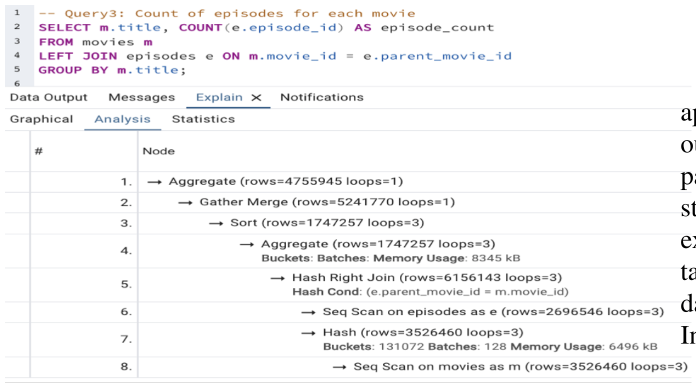
C. *Query 3: Count of Episodes for Each Movie*

1) *Problematic Aspects:*

- Very high loop count for Seq Scans and Aggregate functions.
- The use of LEFT JOIN might be including a lot of NULL entries inflating the loop count.

2) *Performance Improvements:*

- Ensure there are indexes on `movie_id` and `parent_movie_id`.
- Analyze the need for LEFT JOIN; consider INNER JOIN if there are always matching entries.
- Use a more efficient grouping strategy, possibly with subqueries.



```
-- Query3: Count of episodes for each movie
SELECT m.title, COUNT(e.episode_id) AS episode_count
FROM movies m
LEFT JOIN episodes e ON m.movie_id = e.parent_movie_id
GROUP BY m.title;
```

```
#   Node
1.  → Aggregate (rows=4755945 loops=1)
2.      → Gather Merge (rows=5241770 loops=1)
3.          → Sort (rows=1747257 loops=3)
4.              → Aggregate (rows=1747257 loops=3)
                  Buckets: Batches: Memory Usage: 8345 kB
5.                  → Hash Right Join (rows=6156143 loops=3)
                      Hash Cond: (e.parent_movie_id = m.movie_id)
6.                      → Seq Scan on episodes as e (rows=2696546 loops=3)
7.                      → Hash (rows=3526460 loops=3)
                          Buckets: 131072 Batches: 128 Memory Usage: 6496 kB
8.                          → Seq Scan on movies as m (rows=3526460 loops=3)
```

VII. Demo Video and Database Files

A demo video illustrating the features and functionalities of the database system is attached in the zip folder, along with all data files and database scripts (`Create.sql`, `Load.sql`).

VIII. Conclusion

A. *Milestone Achievements*

Throughout the development of the ”Cinematic Insight Odyssey” database, our team achieved several milestones that have established a solid foundation for insightful cinematic data analysis. We successfully transformed complex raw data from IMDB into a structured relational database. We streamlined the schema to ensure normalization to BCNF, facilitating efficient and reliable queries. Our E/R diagram evolved from inception to reflect a refined database schema, where relationships are more accurately represented.

B. *Lessons Learned*

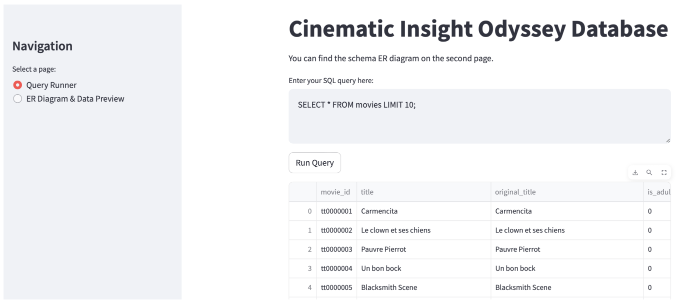
The project reinforced the importance of a robust design phase, especially in complex databases handling vast datasets. We learned that careful planning of data types and relationships significantly enhances query performance. Applying indexing strategies was a practical lesson in optimizing data retrieval. Our journey also highlighted the significance of teamwork and clear role distribution, which was critical for the project's success.

C. *Future Enhancements*

- **Scalability:** Ensure the database can scale to accommodate the ever-growing real-time data from IMDB.
- **Performance:** Continuous optimization strategies, such as more advanced indexing and use of partitioning, can be explored to maintain query performance.
- **Integration:** We plan to integrate the database with front-end applications to provide an interactive UI.
- **Analytics:** Implementing machine learning for predictive analytics and personalized content recommendations.
- **Data Enrichment:** Incorporating additional datasets to enrich the insights our database can provide, such as social media trends and audience feedback mechanisms.

IX. Bonus Task: Interactive Database Web Interface

For the bonus section, we developed an interactive web application, Cinematic Insight Odyssey, that integrates with our PostgreSQL database. Using Streamlit, we crafted a two-page UI that allows dynamic querying and visualizes data structures, including a comprehensive ER diagram. Users can execute custom SQL queries, view the schema, and preview table data, thereby facilitating direct engagement with the database. The live application can be accessed at Cinematic Insight Web App.



Cinematic Insight Odyssey Database

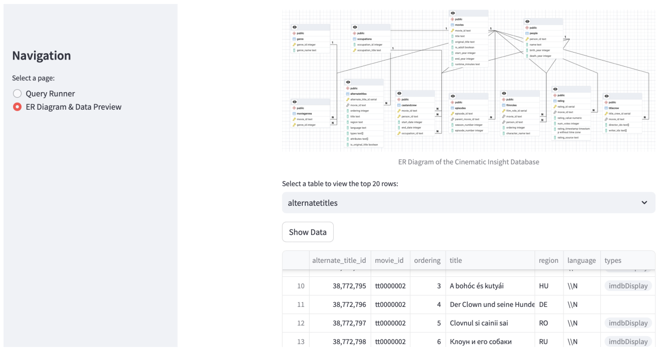
You can find the schema ER diagram on the second page.

Enter your SQL query here:

```
SELECT * FROM movies LIMIT 10;
```

	movie_id	title	original_title	is_adul
0	tt0000001	Carmencita	Carmencita	0
1	tt0000002	Le clown et ses chiens	Le clown et ses chiens	0
2	tt0000003	Pauvre Pierrot	Pauvre Pierrot	0
3	tt0000004	Un bon bock	Un bon bock	0
4	tt0000005	Blacksmith Scene	Blacksmith Scene	0

Fig. 1. Query Interface with Navigation Sidebar



ER Diagram of the Cinematic Insight Database

Select a table to view the top 20 rows: alternatetitles

	alternate_title_id	movie_id	ordering	title	region	language	types
10	38,772,795	tt0000002	3	A bohóc és kutyái	HU	\\N	imdbDisplay
11	38,772,796	tt0000002	4	Der Clown und seine Hunde	DE	\\N	
12	38,772,797	tt0000002	5	Clovnul si cainii sai	RO	\\N	imdbDisplay
13	38,772,798	tt0000002	6	Клоун и его собаки	RU	\\N	imdbDisplay

Fig. 2. ER Diagram & Data Preview Page

X. Team Contributions

A. Pranav Polavarapu (pranavpo):

- Setting direction in terms of Problem Statement, Business insights, Database Design & schema.
- Took charge of the data normalization process, ensuring the database adhered to BCNF principles and contributed to the efficient querying of the database.
- Was responsible for crafting the insightful SQL queries for the project, highlighting the versatility and depth of the database.
- Collaborated on developing the Python scripts used for data transformation and loading for loading accurate data from source files.

B. Sai Keerthi Chelluri (schellur):

- Focused on Design and implementation of the database schema, making pivotal changes on table structures and data relationships.
- Creation & refinement of the final E/R diagram, providing a clear visual representation of the database schema for the team.
- Coordinated the Data Loads into the database, managing the complex process of transforming, manipulating and importing data from various sources.
- Provided key insights for the query performance improvements, regarding strategic indexing and query optimization techniques.

C. Nikhil Raj Yammani (nyammani):

- Responsible for project's data acquisition, sourcing, and preprocessing for integration into the database.
- Managed the debugging and testing of the database using a sample subset, ensuring the integrity and reliability of the data.
- Played a significant role in the documentation and presentation of the database design, detailing the transformation from initial to final schema & methodologies.
- Contributed equally to writing and executing the advanced SQL queries, ensuring a comprehensive demonstration of the database's capabilities.