A Database-Driven Approach for Efficient Movie Data Management for the Film Industry

1st Te Shi
Computer Science and Engineering
University at Buffalo
Buffalo, New York
teshi@buffalo.edu
UBID: teshi
50608573

2nd Jiabao Yao
Computer Science and Engineering
University at Buffalo
Buffalo, New York
jyao27@buffalo.edu
UBID: jyao27
50602483

***Abstract*—This report presents a powerful movie/TV database, including its structure, detailed contents, and a demonstration of its capabilities. The application aims to provide convenience for film production and facilitate information collection for industry professionals, film enthusiasts, data analysts, and industry observers.**

***Index Terms*—Database, Entity-Relationship Diagram, Relational Schema, Normalization, SQL Queries, Triggers, SQL Functions, Indexing, PostgreSQL, Power BI, Film Industry, Movie Data Management**

I. Problem Statement

The purpose of this project is to develop a database-driven application that serves as a powerful decision-making tool for film production, provides insights into the movie industry, and functions as an information query platform for cinephiles. This general-purpose database platform will enable structured data analysis, making it a valuable resource for industry professionals, researchers, and film enthusiasts alike.

Film and TV production and industry analysis are core aspects of the film and TV business. A successful project represents not only a profitable investment for stakeholders but also a major cultural achievement that can shape societies and global perspectives. However, film production is also highly risky—poorly managed decisions can result in financial losses, unemployment, and setbacks to a country's cultural landscape. Similarly, analyzing and observing industry trends is as crucial as in any other sector, given the significant influence of film and television media.

Our database product serves as a valuable tool for decision-making in film production. It provides quantified data for professionals and industry observers, enabling informed choices. For example, when casting for a film, key considerations include: Is this actor a good fit for the genre? What is their past reputation in similar roles? How well does the director collaborate with them? Will the film perform well in specific regions? Our database effectively addresses these questions, offering data-driven insights to support better decision making.

Current popular movie databases, such as IMDb and Letterboxd, provide extensive and comprehensive information about films and TV shows. However, these platforms lack easy-to-use functions for quantifying data and revealing relationships between different entities in filmmaking. As a result, they are not highly effective tools for decision-making in film production or in-depth industry analysis. Also, those platforms are primarily used for browsing and viewing. There is no established platform that allows users to freely explore the dataset.

A simple spreadsheet solution, such as Excel, is obvisouly inadequate for this type of data-intensive project for several reasons. Firstly Excel struggles with large datasets. The whole database involved in this project spans multiple interconnected datasets, each potentially containing millions of records. Excel is not designed to handle such extensive data efficiently, and as the dataset grows, spreadsheet operations become slow and impractical. Complex queries involving multiple datasets would also become sluggish, making real-time analysis difficult. By comparison, the powerful Data Manipulation Language SQL supports can complete the job efficiently and easily.

Furthermore, Excel lacks robust constraints, data validation mechanisms, and referential integrity enforcement, which increases the risk of data inconsistencies. Unlike a relational database that ensures accuracy through primary and foreign keys, Excel relies on manual data entry, making it prone to duplication and errors.

Additionally, a well-designed database is more adaptable to future development needs. It can integrate with other applications such as web application or machine learning development, support real-time analytics, and serve as the foundation for advanced functionalities, such as recommendation systems or predictive modeling. Excel, in contrast, lacks the flexibility and scalability required for such expansions, making it unsuitable for long-term use in this project.

Unlike Excel-based project, our project will develop a database system that decomposes the existing open-source IMDb dataset into approximately 14 interrelated schemes, based on Entity-Relationship (E-R) and following strictly with relational database design principles to establish meaningful connections between different data entities. Our dataset comprises millions of records, ensuring the accuracy and reliability of our project.

In summary, our relational database, with millions of records, goes beyond typical movie database platforms. It serves as a powerful tool for film production, industry analysis, and film education and research.

II. Target User Group

A. Film Producer and Project Management Professionals

The project serves as a powerful decision-making tool for film production. By querying the database, film professionals can obtain valuable insights into film-related information, helping them make well-informed decisions. Potential applications include assessing the past reputation of film personnel within specific genres, analyzing the synergy between different cast combinations to validate a project's success rate, and evaluating the popularity of certain intellectual properties, genres, directors, or actors within specific regions or time periods. Such insights can assist in making strategic distribution and production decisions.

For instance, the actor evaluation problem mentioned earlier can be easily and efficiently addressed using our database. The actor schema is linked to a general movie schema, which is further connected to separate schemas for genre and ratings information. This structured approach enables seamless queries to assess an actor's past performance in comedy films, providing valuable insights for casting decisions.

B. Industry Analyst and Observers

Our project also provides valuable tools for industry analysts and observers. Trends in the film industry are often reflected through newly released movies, and our database enables analysts to efficiently capture and analyze these patterns. By leveraging structured data, analysts can identify emerging trends, shifts in audience preferences, and industry developments. These insights can be highly beneficial for investors and individuals looking to enter the film industry, helping them make informed and strategic decisions.

C. Movie enthusiasts and Educator

By leveraging the relational database, movie enthusiasts and educators can uncover fascinating insights about the film industry. For example, they can explore which director Leonardo DiCaprio has collaborated with the most or identify the highest-rated Chinese film among American audiences. These types of queries can serve both educational purposes and casual exploration, allowing users to engage with movie data in an interactive and informative way.

D. Programming Developer and AI Engineer

The database can also serve as the foundation for web applications and AI model training. For instance, a visualized platform could be developed to allow users without SQL knowledge to easily access and explore movie-related data. Additionally, the structured dataset could be used to build a recommendation system, enabling exploratory data analysis and machine learning training to predict user preferences, suggest films, or analyze industry trends.

E. Administer of the Database

The database project should be managed by people with professional knowledge of database and the domain knowledge of the film industry who can ensure the integrity, security and performance of the database. Specifically a professional data analysis team could be the administer which can be responsible for mainlining the database, uploading new data and ensure data accuracy. The project is designed to be easily scalable and expandable, allowing for continuous data enrichment. New information can be incorporated seamlessly, ensuring the database remains relevant and comprehensive. For example, the rating schema can be expanded to include ratings from multiple platforms, providing a more diverse and accurate representation of audience reception. Additionally, a box office schema could be integrated and linked to the general TV and Movie schema, enabling analysis of a film's financial performance alongside other attributes. These enhancements ensure the database remains adaptable to evolving industry needs.

III. E/R Diagram

A. Entity-Relation Diagram

To improve the efficiency and structure of our database, we divided the original dataset into multiple tables based on design principle of relational database and the requirements of our project. Thirteen relations are included in E/R diagram, as shown in Figure 1, which are *Genres*, *TV_Movie*, *Genres_Movie*, *Episode*, *Episode_Details*, *Title_Aliases*, *Cast*, *Profession*, *Crew*, *People*, *Profession_People*, *Ratings*, and *Famous_Work*.

B. Entity Description

The following is a brief description of each relation in the E/R diagram:

- *TV_Movie* contains basic information about movies and TV shows (excluding episodes).
- *Title_Aliases* stores alternative titles for TV shows and movies.
- *Episode_Details* stores episode-specific metadata.
- *Episode* stores basic information about TV episodes, separate from *TV_Movie*.
- *Genres* contains unique genre classifications.
- *Genres_Movies* establishes the many-to-many relationship between *TV_Movie* and *Genres*.
- *People* contains details of individuals in the database.
- *Profession* stores unique profession names.
- *Profession_People* links individuals to their professions.
- *Crew* contains all related people for movie or TV record in the database except actors.
- *Famous_Work* connects people to their notable works.
- *Ratings* stores rating details for each movie or TV show.
- *Cast* establishes the many-to-many relationship between *TV_Movie* and *People*.

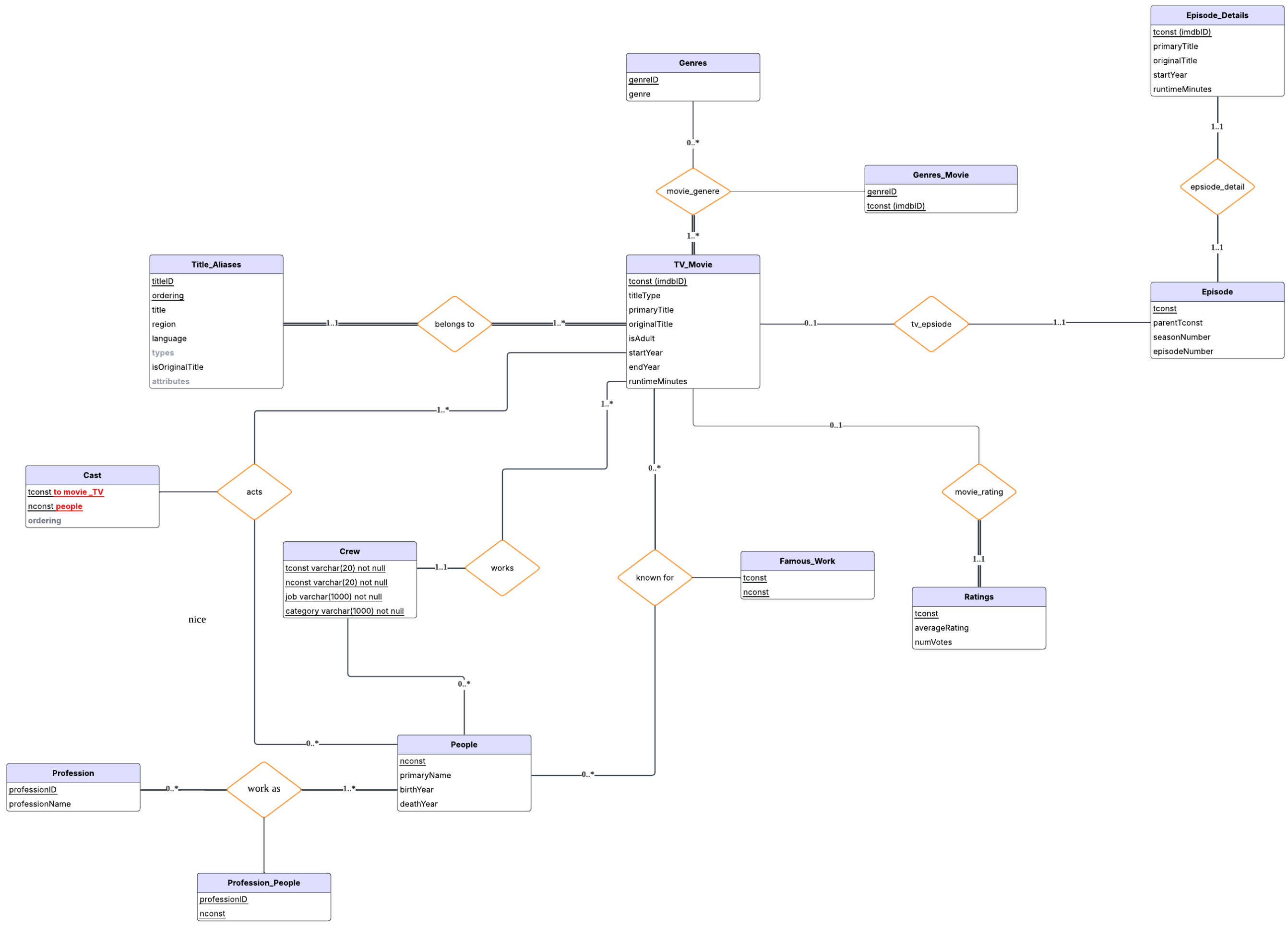


Fig. 1. ER Diagram

C. Relation Description

As shown in the ER Diagram, the relationships between the tables are also illustrated. A detailed explanation is provided below.

- Each movie could have one or many genres. The *TV_Movie* table has a many-to-many relationship with *Genres* through the *Genres_Movies* relationship.
- The episodes for each TV series is stored separately in *Episodes*. Therefore, *TV_Movie* has a one-to-many relations with *Episodes* since each TV series consists of multiple episodes.
- Each TV episode will have detailed information stored in *Episode_Details*. So *Episodes* has a one-to-one relationship with *Episode_Details*.
- The *Title_Aliases* table stores alternative titles for movies and TV shows. Each movie can have more than one titles so, establishing a many-to-one relationship with *TV_Movie*.
- For actors/actress, they will have a zero-to-many relationship with *TV_Movie* via *Cast* since one actor can appear in many different products.
- For other professional people, they will also have a zero-to-many relationship with *TV_Movie* via *Crew* which states their job type.
- The *People* has a one-to-many relationship with *TV_Movie* via *famous_work* since one person can have more than one famous work.
- The *People* table has a one-to-many relationship with *Profession* through *Profession_People*, representing the multiple roles a film professional can have.
- The *TV_Movie* table has a one-to-one relationship with *Ratings* since one product will only have one rating in the IMDB dataset.

IV. Relation Schema Details

A. Relation 1: TV_Movie

- **tconst**: Unique identification for each movie or TV. `VARCAHR(20),NOT NULL`

- **titleType**: Indicates the type of the production. For example, it can be TV series, movie, or short film, among others. `VARCHAR(50), NULL`
- **primaryTitle**: The title the product is widely known as `VARCHAR(200), NULL`
- **originalTitle**: The original title of this product `VARCHAR(200), NULL`
- **isAdult**: Indicates whether the content is intended for adults only. `BOOLEAN, 0` (since most of the product is not just for adult.)
- **startYear**: The release year of the product `INTEGER, NULL`
- **endYear**: The end year of the product. Only for TV series `INTEGER, NULL`
- **runtimeMinutes**: the length of the product. `INTEGER, NULL`
- **Primary Key**:`(tconst)` It is an unique identification for each product in IMDB dataset.

B. Relation 2: Genres

- **genreID**: Unique identifier for each genre. `INTEGER, NOT NULL`
- **genre**: The name of the genre (e.g., Action, Drama, Comedy). `VARCHAR(20), NOT NULL`
- **Primary Key**:`(genreID)` It uniquely identifies each genre.

C. Relation 3: Genres_Movie

- **genreID**: Unique identifier for each genre. `INT NOT NULL`
- **genre**: The name of the genre (e.g., Action, Comedy, Drama). `VARCHAR(20) NOT NULL UNIQUE`
- **Primary Key**: `(genreID, tconst)` The combination of genre and movie must be unique.
- **Foreign Keys**:
  - `tconst` → *TV_Movie(tconst)*
  - `genreID` → *Genres(genreID)*

D. Relation 4: Episode

- **tconst**: Unique identifier for each episode. `VARCHAR(20), NOT NULL`
- **parentTconst**: Foreign key referencing `TV_Movie(tconst)`, indicating the parent show. `INTEGER, NOT NULL`
- **seasonNumber**: The season in which the episode appears. `INTEGER, NOT NULL`
- **episodeNumber**: The specific episode number within the season. `INTEGER, NOT NULL`
- **Primary Key**: `(tconst)` It uniquely identifies each episode.

E. Relation 5: Episode_Details

- **tconst**: `Foreign key` referencing `Episode(tconst)`, linking episode details to their corresponding episodes. `VARCHAR(20), NOT NULL`
- **primaryTitle**: The title by which the episode is widely known. `VARCHAR(200), NOT NULL`
- **originalTitle**: The original title of the episode. `VARCHAR(200), NOT NULL`
- **startYear**: The release year of the episode. `INTEGER, NOT NULL`
- **runtimeMinutes**: The runtime duration of the episode in minutes. `INTEGER, NULL`
- **Primary Key**:`(tconst)` It uniquely identifies each episode's details.

F. Relation 6: Title_Aliases

- **titleID**: Unique identifier for each movie or TV show. `VARCHAR(20) NOT NULL`
- **ordering**: Index to distinguish different title aliases for the same movie or TV show. `INT NOT NULL`
- **region**: The region associated with each title alias. `VARCHAR(20) NOT NULL`
- **language**: The language associated with each title alias. `VARCHAR(20) NOT NULL`
- **isOriginalTitle**: Indicates whether the alias is the original title. `BOOLEAN NOT NULL`
- **Primary Key**: (`titleID`, `ordering`) The composite key uniquely identifies each title alias for a movie or TV show.
- **Foreign Keys**:
  - `titleID` → *TV_Movie(tconst)*

G. Relation 7: Cast

- **tconst**: Unique identifier for each movie or TV show. `VARCHAR(20) NOT NULL`
- **nconst**: Unique identifier for each person. `VARCHAR(20) NOT NULL`
- **Primary Key**: (`tconst`, `nconst`) This composite key ensures each person is uniquely linked to a movie or TV show.
- **Foreign Keys**:
  - `tconst` → *TV_Movie(tconst)*
  - `nconst` → *People(nconst)*

H. Relation 8: Crew

- **tconst**: Unique identifier for each movie or TV show. `VARCHAR(20) NOT NULL`
- **nconst**: Unique identifier for each person. `VARCHAR(20) NOT NULL`
- **job**: Specific job title assigned to a person for a movie or TV show. `VARCHAR(50) NOT NULL`
- **category**: The broad classification of jobs related to movies or TV shows. (e.g. Director, Actor, Cinematographer) `VARCHAR(50) NOT NULL`
- **job**: Specific job title within a category. For example, for category director, the job can be director, assistance director, casting director. `VARCHAR(50) NOT NULL`
- **Primary Key**: (`tconst`, `nconst`, `job`, `category`) This composite key ensures to identify every job for every movie for every person.

- **Foreign Keys**:
  - `tconst` → *TV_Movie(tconst)*
  - `nconst` → *People(nconst)*

I. Relation 10: People

- **nconst**: Unique identifier for each person. `VARCHAR(20) NOT NULL`
- **primaryName**: The full name of the person. `VARCHAR(200) NOT NULL`
- **birthYear**: The birth year of the person. `INT NULL`
- **deathYear**: The death year of the person (if applicable). `INT NULL`
- **Primary Key**: (`nconst`)

J. Relation 11: Profession_People

- **professionID**: Unique identifier for each profession. `INT NOT NULL`
- **nconst**: Unique identifier for each person. `VARCHAR(20) NOT NULL`
- **Primary Key**: (`profession_id`, `nconst`) This composite key ensures that a person can have multiple professions but must be uniquely associated with them.
- **Foreign Keys**:
  - `professionID` → *Profession(professionID)*
  - `nconst` → *People(nconst)*

K. Relation 12: Profession

- **professionID**: Unique identifier for each profession. `INT NOT NULL`
- **professionName**: The name of the profession. `VARCHAR(50) NOT NULL`
- **Primary Key**: (`professionID`) This ensures that each profession has a unique identifier.

L. Relation 13: Famous_Work

- **tconst**: Unique identifier for each movie or TV show. `VARCHAR(20) NOT NULL`
- **nconst**: Unique identifier for each person. `VARCHAR(20) NOT NULL`
- **Primary Key**: (`tconst`, `nconst`) This composite key ensures that a person is uniquely linked to a famous work.
- **Foreign Keys**:
  - `tconst` → *TV_Movie(tconst)*
  - `nconst` → *People(nconst)*

M. Relation 14: Ratings

- **tconst**: Unique identifier for each movie or TV show. `VARCHAR(20) NOT NULL`
- **averageRating**: The average rating given to the movie or TV show. `FLOAT NULL`
- **numVotes**: The total number of votes received for the rating. `INT NOT NULL`
- **Primary Key**: (`tconst`) This ensures that each movie or TV show has a unique rating entry.
- **Foreign Key**:
  - `tconst` → *TV_Movie(tconst)*

Delete cascade is enforced for all foreign keys to ensure that when a referenced record is deleted, all related records in dependent tables are also removed, maintaining data integrity.

V. BCNF Verification

A. Relations and Their Functional Dependencies (FDs)

A list of relations and their functional dependencies (FDs) are shown below. To verify whether a relation is in BCNF, we must ensure that for every FD X → Y, X is a superkey or the dependency is trivial.

B. Relations and Their Functional Dependencies (FDs)

1) Genres(genreID, genre)
   FDs: genreID → genre
   BCNF Verification: genreID is a candidate key and determines all attributes, so Genres is in BCNF.
2) Genres_Movie(genreID, tconst)
   BCNF Verification: The composite key (genreID, tconst) determines all attributes, so Genres_Movie is in BCNF.
3) Episode(tconst, parentTconst, seasonNumber, episodeNumber)
   FDs: tconst → parentTconst, seasonNumber, episodeNumber
   BCNF Verification: tconst is a candidate key and determines all attributes, so Episode is in BCNF.
4) Episode_Details(tconst, primaryTitle, originalTitle, startYear, runtimeMinutes)
   FDs: tconst → primaryTitle, originalTitle, startYear, runtimeMinutes
   BCNF Verification: tconst is a candidate key and determines all attributes, so Episode_Details is in BCNF.
5) TV_Movie(tconst, titleType, primaryTitle, originalTitle, isAdult, startYear, endYear, runtimeMinutes)
   FDs: tconst → titleType, primaryTitle, originalTitle, isAdult, startYear, endYear, runtimeMinutes
   BCNF Verification: tconst is a candidate key and determines all attributes, so TV_Movie is in BCNF.
6) Ratings(tconst, averageRating, numVotes)
   FDs: tconst → averageRating, numVotes
   BCNF Verification: tconst is a candidate key and determines all attributes, so Ratings is in BCNF.
7) Famous_Work(tconst, nconst)
   FDs: tconst, nconst → tconst, nconst (trivial FD)
   BCNF Verification: Since this is a trivial dependency, Famous_Work is in BCNF.
8) Title_Aliases(titleID, ordering, title, region, language, isOriginalTitle)
   FDs: titleID, ordering → title, region, language, isOriginalTitle
   BCNF Verification: (titleID, ordering) is a candidate key, so Title_Aliases is in BCNF.
9) Profession(professionID, professionName)
   FDs: professionID → professionName
   BCNF Verification: professionID is a candidate key and determines all attributes, so Profession is in BCNF.

10) People(nconst, primaryName, birthYear, deathYear)
FDs: nconst $\rightarrow$ primaryName, birthYear, deathYear
BCNF Verification: nconst is a candidate key and determines all attributes, so People is in BCNF.
11) Profession_People(professionID, nconst)
FDs: professionID, nconst $\rightarrow$ professionID, nconst (trivial FD)
BCNF Verification: Since this is a trivial dependency, Profession_People is in BCNF.
12) Cast(tconst, nconst)
FDs: tconst, nconst $\rightarrow$ tconst, nconst (trivial FD)
BCNF Verification: Since this is a trivial dependency, Cast is in BCNF.
13) Crew(tconst, nconst, category, job)
FDs:
- tconst, nconst, category, job $\rightarrow$ tconst, nconst, category, job
BCNF Verification: Since this is a trivial dependency, Crew is in BCNF.

Final Conclusion: All relations are naturally in BCNF.

VI. SQL Query Test

In this section, around 15 PostgreSQL queries are tested, and the results are presented. The purpose is to verify the functionality of our database. Many of these queries are also powerful and will be used later in the visualization phase to extract meaningful information that showcases our project's capabilities.

A. Insertion 1 - Insert TV or Movie Information

This query inserts new records related to TV or movie information.

```
INSERT INTO tv_movie
VALUES ('tt0111161','movie','The Shawshank
    Redemption','The Shawshank Redemption
   ', true, 1994, NULL, 142)
```

Name	Value
Updated Rows	1
Execute time	0.021s
Start time	Fri Apr 25 22:10:24 EDT 2025
Finish time	Fri Apr 25 22:10:24 EDT 2025
Query	INSERT INTO tv_movie
	VALUES ('tt0111161','movie','The Shawshank Redemption','The Shawshank Redemption', true, 1994, NULL, 142)

Fig. 2. Insertion 1

B. Insertion 2 - Insert People

This query inserts new records related to filmmaker information

```
INSERT INTO people
VALUES ('nm10025009','Yiwen Zhu', 1993,
   NULL)
```

Name	Value
Updated Rows	1
Execute time	0.011s
Start time	Fri Apr 25 22:19:18 EDT 2025
Finish time	Fri Apr 25 22:19:18 EDT 2025
Query	INSERT INTO people
	VALUES ('nm10025009','Yiwen Zhu', 1993, NULL)

Fig. 3. Insertion 2

C. Deletion 1 - Delete TV/Movie Record

This query deletes a TV or movie record. It not only removes the corresponding entry but also deletes all related records in other tables that reference it via foreign keys.

```
DELETE FROM tv_movie
WHERE tconst = 'tt0111161'
```

	tconst	titletype	primarytitle	originaltitle	isadult	startyear	endyear	runtimeminutes
1	tt34956443	movie	Ne Zha 2	Nezha: Mo tong nao hai	[ ]	2,025	[NULL]	143

Fig. 4. Movie record before delete is executed

	tconst	nconst	job	category
1	tt34956443	nm11088223	producer	producer
2	tt34956443	nm1169562	script	writer
3	tt34956443	nm17018329	"loosely based on the novel ""Investiture of the Gods"" by"	writer
4	tt34956443	nm8358501	"loosely based on the novel ""Investiture of the Gods"" by"	writer

Fig. 5. Crew information before the movie is deleted

	tconst	titletype	primarytitle	originaltitle	isadult	startyear	endyear	runtimeminutes

Fig. 6. Movie Record after the movie is deleted

	tconst	nconst	job	category

Fig. 7. Crew information after the movie is deleted

D. Deletion 2 - Delete Episode

This query deletes a record in episode relation and also all related records in other table that reference it via foreign keys.

```
DELETE FROM episode
WHERE tconst = 'tt0612823'
```

Statistics 1

Name	Value
Updated Rows	1
Execute time	0.023s
Start time	Fri Apr 11 18:17:33 EDT 2025
Finish time	Fri Apr 11 18:17:33 EDT 2025
Query	delete from episode where tconst = 'tt0612823'

Fig. 8. Delete 2

E. Update 1 - Update Alternative Title

This query update the alternative title of a movie to a new one.

```
UPDATE title_aliases
SET "title" = '克隆战争'
WHERE titleId = 'tt0361243' AND "ordering"
= 35
```

titleid	ordering	title	region	language	isoriginaltitle
tt0361243	17	Star Wars: Clone Wars	TR	tr	[ ]
tt0361243	18	Star Wars: Clone Wars	US	[NULL]	[ ]
tt0361243	19	Star Wars: Clone Wars	ZA	en	[ ]
tt0361243	20	Csillagok háborúja: Klónok háborúja	HU	[NULL]	[ ]
tt0361243	21	Gwiezdne wojny: Wojny klonów	PL	[NULL]	[ ]
tt0361243	22	Ratovi zvijezda: Ratovi klonova	HR	[NULL]	[ ]
tt0361243	23	Razboiul Stelelor: Razboiul clonelor	RO	[NULL]	[ ]
tt0361243	24	Star Wars Vintage: Clone Wars 2D Mi	XWW	en	[ ]
tt0361243	25	Star Wars: Guerras clónicas	MX	[NULL]	[ ]
tt0361243	26	Star Wars: Las Guerras Clon	ES	[NULL]	[ ]
tt0361243	27	Ο πόλεμος των άστρων: Πόλεμος τι	GR	[NULL]	[ ]
tt0361243	28	Жұлдызды соғыстар: Клондық соғы	KZ	[NULL]	[ ]
tt0361243	29	Звездани ратови: Ратови клонова	RS	[NULL]	[ ]
tt0361243	30	Звездные войны: Войны Клонов	RU	[NULL]	[ ]
tt0361243	31	Звёздные войны: Войны Клонов	RU	[NULL]	[ ]
tt0361243	32	Ратови звезда: Ратови клонова	RS	[NULL]	[ ]
tt0361243	33	स्टार वार्स: क्लोन वार्स	IN	hi	[ ]
tt0361243	34	スター・ウォーズ クローン大戦	JP	ja	[ ]
tt0361243	35	星際大戰：複製人之戰	TW	[NULL]	[ ]

Fig. 9. Update 1: before the title is updated

```
tt0361243|        29|Звездани ратови: Ратови клонова
tt0361243|        30|Звездные войны: Войны Клонов
tt0361243|        31|Звёздные войны: Войны Клонов
tt0361243|        32|Ратови звезда: Ратови клонова
tt0361243|        33|स्टार वार्स: क्लोन वार्स                    |II
tt0361243|        34|スター・ウォーズ クローン大戦
tt0361243|        35|克隆戰爭
```

Fig. 10. Update 1: after the title is updated

F. Update 2 - Update Ratings

This query updates the rating of a movie or TV. In the example, the rating and number of votes were changed from 8.6 and 17 to 8 and 19 respectively.

```
Update ratings
SET averageRating = 8.0, numVotes = 19
WHERE tconst = 'tt35957962'
```

	tconst	averagerating	numvotes
1	tt35957962	8.6	17

Fig. 11. Update 2: before the rating is updated

	tconst	averagerating	numvotes
1	tt35957962	8	19

Fig. 12. Update 2: after the rating is updated

G. Query using Join – Get Actor/Actress's Filmography

This query extract an actor or actress's filmography, which are an important information when making casting decision. In the example below, it displays the actor Aachi Manorama's all works.

```
select primaryname, primarytitle
from people p
natural join "Cast" c
natural join tv_movie tm
where p.primaryname = 'Aachi Manorama'
```

	primaryname	primarytitle
1	Aachi Manorama	Kanna Nalama
2	Aachi Manorama	Vaazhvey Maayam
3	Aachi Manorama	Engal Thangam
4	Aachi Manorama	Melnattu Marumagal
5	Aachi Manorama	Thiruvilayadal
6	Aachi Manorama	Marumagaley Varuga
7	Aachi Manorama	Thaaye Unakkaga!
8	Aachi Manorama	Kuppathu Raja
9	Aachi Manorama	Andru Kanda Mugam
10	Aachi Manorama	Galatta Kalyanam
11	Aachi Manorama	Nattupura Pattu
12	Aachi Manorama	Unnai Vaazhthi Paadukiren
13	Aachi Manorama	Paaru Paaru Pattinam Paaru
14	Aachi Manorama	Uyarnthavargal
15	Aachi Manorama	Saamy
16	Aachi Manorama	Naan Sollum Rahasiyam
17	Aachi Manorama	Perazhagan
18	Aachi Manorama	Andipatti Arasanpatti
19	Aachi Manorama	Kizhvanam Sivakkam
20	Aachi Manorama	Raja Part Rangadurai
21	Aachi Manorama	The Dance of Life
22	Aachi Manorama	Saranalayam

Fig. 13. An actor/actress's filmography

H. Query using Order By – Top 10 Greatest Movies of All Time

This query extract the 10 best movies in the film history based on the rating and viewer information from our database.

```
SELECT tm.primarytitle, r.averagerating
FROM tv_movie tm
natural join ratings r
where r.numvotes > 10000 and (titleType =
    'movie' or titleType = 'tvMovie')
order by r.averagerating desc
limit 10
```

	primarytitle	averagerating
1	12 Angry Men	9
2	The Dark Knight	9
3	Pulp Fiction	8.9
4	The Chaos Class Failed the Class	8.8
5	Shoah	8.7
6	Soorarai Pottru	8.7
7	Daman	8.7
8	Se7en	8.6
9	Zübük	8.5
10	Alien	8.5

Fig. 14. Top 10 Movies

I. Query using Group By – Find Movie/TV Statistics Based On Genre

This query counts the number of films and TV shows in our database by genre. It helps users gain a better overall understanding of the content distribution in the database.

```sql
SELECT g.genre, COUNT(*) AS movie_count
FROM tv_movie tm
NATURAL JOIN genres_movie gm
NATURAL JOIN genres g
GROUP BY g.genre
ORDER BY movie_count DESC
```

	genre	movie_count
1	Drama	36,206
2	Documentary	23,520
3	Comedy	21,973
4	Action	7,649
5	Romance	7,214
6	Crime	6,175
7	Thriller	6,089
8	Horror	5,098
9	Adventure	4,597
10	Family	4,537
11	Music	3,651
12	Animation	3,263
13	Mystery	3,024
14	Reality-TV	2,956
15	Biography	2,915
16	History	2,835
17	Fantasy	2,701
18	Talk-Show	2,540
19	Sci-Fi	2,338
20	Sport	1,901
21	Musical	1,615
22	Adult	1,237
23	War	1,210
24	News	1,172

Fig. 15. Number of Works for Each Genres

J. Query using Sub-query – Professionals Specializing in Specific Genres

This query retrieves the top professionals for a specific genre, based on their job category and the average ratings of their works within that genre. It helps users quickly identify the best team for a film or TV project.

```sql
select dir_work.primaryname, diravg
from (select p.nconst, p.primaryname, avg(
    r.averagerating) as dirAvg
from crew c
natural join tv_movie tm
natural join genres_movie gm
natural join genres g
natural join ratings r
natural join people p
where c.category = 'director' and g.genre
    = 'Comedy' and p.deathYear is not NULL
group by p.nconst, p.primaryname
order by dirAvg desc) as dir_work
where dir_work.diravg > (select AVG(r.
    averagerating)
from tv_movie tm
natural join ratings r
natural join genres g
natural join genres_movie gm
where g.genre = 'Comedy')
```

	primaryname	diravg
1	Víctor Manuel Castro	8.6
2	Hans Mahler	8.2
3	Menios Ditsas	8.2
4	Petar B. Vasilev	8.1
5	Fred C. Newmeyer	8.1
6	Rudolf Noelte	8.1
7	David Cherkasskiy	8.1
8	Rasim Ojagov	7.9
9	Ladislao Vajda	7.9
10	George Bloomfield	7.9
11	George Lessey	7.8
12	Howard Hawks	7.8
13	Siddique	7.75
14	John G. Blystone	7.7
15	Pál Fejös	7.7

Fig. 16. Find Good Professionals

K. Other Queries 1 – Find Movie/TV Statistics Based On Genre

This query is a modification of Query I. In addition to the number of works, it also summarizes the average rating and number of votes by genre. It serves as an overview of the database for users.

```sql
SELECT g.genre, COUNT(*) AS movie_count,
    ROUND(AVG(r.averagerating)::numeric,
    1) as avg_ratings, ROUND(AVG(r.
    numvotes )::numeric) as avg_viewer
FROM tv_movie tm
NATURAL JOIN genres_movie gm
NATURAL JOIN genres g
natural join ratings r
GROUP BY g.genre
ORDER BY movie_count DESC
```

genre	movie_count	avg_ratings	avg_viewer
Drama	20025	6.4	3848
Comedy	12411	6.2	3513
Documentary	8391	7.2	309
Romance	4911	6.2	3313
Action	4344	5.9	9661
Crime	4195	6.2	8924
Thriller	3435	5.7	6539
Adventure	2921	6.2	10388
Horror	2836	5.1	4839

Fig. 17. Database Statistics by Genre

L. Other Queries 2 – Genre Popularity over Time

This query groups film and TV titles by genre and release decade, showing the average popularity of each group based on average rating and number of viewers.

```sql
SELECT
  g.genre,
  (startyear / 10) * 10 AS decade,
  ROUND(AVG(r.averagerating )::numeric, 1)
      AS avg_rating,
  ROUND(AVG(r.numvotes)::numeric) AS
     avg_view_count
```

```
FROM tv_movie tm
NATURAL JOIN genres_movie gm
NATURAL JOIN genres g
NATURAL JOIN ratings r
WHERE startyear IS NOT NULL AND startyear
    >= 1900
GROUP BY g.genre, decade
ORDER BY decade, g.genre;
```

genre	decade	avg_rating	avg_view_count
Comedy	1900	4.6	13
Documentary	1900	4.6	20
Drama	1900	2.2	12000
Music	1900	3.4	30
Musical	1900	4.6	13
News	1900	5.3	23
Sport	1900	5.3	23
Action	1910	5.6	212
Adventure	1910	5.1	29
Biography	1910	6.9	232
Comedy	1910	6.1	131
Crime	1910	6.0	122
Documentary	1910	5.5	16

Fig. 18. Change of Popularity of Movie/TV Over Time

M. Function 1 – Genre Popularity over Time

This function returns the top professionals in a specific genre and role. A professional is considered superior if the average rating of their works exceeds the overall average rating for that genre. In the example below, I select superior director for drama movies.

```
create function get_professionals_by_genre
    (target_genre VARCHAR(100),
    target_category VARCHAR(1000))
returns table(category VARCHAR(1000),
    genre VARCHAR(100), primaryname
    VARCHAR(1000), avg_score float) as $$
begin
    return QUERY
    select target_category as category, g.
        genre, p.primaryname, avg(r.
        averageRating) as avg_score
    from crew c
    natural join tv_movie tm
    natural join genres_movie gm
    natural join genres g
    natural join ratings r
    natural join people p
    where c.category = target_category and
        g.genre = target_genre and p.
        deathYear is not NULL
    group by p.nconst, g.genre, p.
        primaryname
    having avg(r.averageRating) > (
        select AVG(r2.averagerating)
        from tv_movie tm2
        natural join ratings r2
        natural join genres g2
        natural join genres_movie gm2
        where g2.genre = target_genre
    );
end;
$$ language plpgsql

SELECT *
FROM get_professionals_by_genre('Drama', '
    director');
```

category	genre	primaryname	avg_score
director	Drama	Luis Buñuel	7.6
director	Drama	Fritz Lang	7.0
director	Drama	Theodoros Angelopoulos	7.9
director	Drama	Frank Capra	6.7
director	Drama	Cecil B. DeMille	6.6
director	Drama	Alain Delon	7.2
director	Drama	Rainer Werner Fassbinder	7.2
director	Drama	Howard Hawks	7.8
director	Drama	Sidney Lumet	6.9
director	Drama	Sam Peckinpah	7.2
director	Drama	Giuliano Carnimeo	6.5
director	Drama	Fred Zinnemann	7.8
director	Drama	Derwin Abrahams	6.7

Fig. 19. Top Directors for Comedy Works

N. Function 2 – Top Film/TV Based on Title Type and Genre

Retrieving top films based on specific standards is a common use case for a movie and TV database. Therefore, this query is implemented as a function to allow more flexible use. The function uses an IF-ELSE block to select results based on different standards, and the selection criteria can be further extended based on actual needs.

```
CREATE OR REPLACE FUNCTION
    top_works_by_genre(
    input_type VARCHAR(50),
    input_genre VARCHAR(100),
    limit_cnt INT
)
RETURNS TABLE(title VARCHAR(1000), rating
    FLOAT) AS $$
BEGIN
    IF input_type = 'movie' THEN
        RETURN QUERY
        SELECT tm.primarytitle, r.
            averagerating
        FROM tv_movie tm
        NATURAL JOIN ratings r
        NATURAL JOIN genres_movie gm
        NATURAL JOIN genres g
        WHERE r.numvotes > 10000
          AND (tm.titletype = 'movie' OR tm.
            titletype = 'tvMovie')
          AND g.genre = input_genre
        ORDER BY r.averagerating DESC
        LIMIT limit_cnt;

    ELSIF input_type = 'tv' THEN
        RETURN QUERY
```

```
        SELECT tm.primarytitle, r.
           averagerating
        FROM tv_movie tm
        NATURAL JOIN ratings r
        NATURAL JOIN genres_movie gm
        NATURAL JOIN genres g
        WHERE r.numvotes > 10000
          AND (tm.titletype = 'tvSeries' OR
             tm.titletype = 'tvMiniSeries')
          AND g.genre = input_genre
        ORDER BY r.averagerating DESC
        LIMIT limit_cnt;

    ELSE
        RETURN QUERY
        SELECT tm.primarytitle, r.
           averagerating
        FROM tv_movie tm
        NATURAL JOIN ratings r
        NATURAL JOIN genres_movie gm
        NATURAL JOIN genres g
        WHERE r.numvotes > 10000
          AND g.genre = input_genre
        ORDER BY r.averagerating DESC
        LIMIT limit_cnt;
    END IF;
END;
$$ LANGUAGE plpgsql;

SELECT *
FROM get_professionals_by_genre('Drama', '
    director');
```

```
title         |rating|
-------------+-------+
Kota Factory|    9.0|
The Office  |    9.0|
Seinfeld    |    8.9|
```

Fig. 20. Top 3 TV Series

O. Function 3 – Update Ratings and Number of Viewers

Ratings and the number of viewers for a work change dynamically, so it is necessary to save the query as a function for convenient and flexible use.

```
CREATE FUNCTION insert_ratings(tt VARCHAR
    (20), averageRatings FLOAT, numVotes
    INT, title VARCHAR(1000))
RETURNS VOID AS $$
DECLARE movie_title VARCHAR(1000);
begin
    select primaryTitle into movie_title
        from tv_movie where tconst = tt
       ;

    if movie_title is null then
         insert into tv_movie(tconst,
            primaryTitle)
         values (tt, title);
    END IF;

    insert into ratings
    values (tt, averageRatings, numVotes
       );

end;
$$ language plpgsql;
```

VII. Transaction Triggers

This section discusses the implementation of transactions to ensure atomic updates across database operations, such as:

- Modifying multiple columns in a single relation,
- Updating multiple tables simultaneously, or
- Performing mixed operations (insertions, deletions, updates) on one or more relations.

A transaction commits only if all constituent operations succeed. For instance, movie ratings require real-time updates to both the average score and vote count. An example transaction is shown below:

```
BEGIN;

UPDATE ratings
SET averageRating = 89,
   numVotes = 12000
WHERE tconst = 'tt9999999';

UPDATE ratings
SET averageRating = 7.9,
   numVotes = 12000
WHERE tconst = 'tt9999999';

COMMIT;
```

	tconst [PK] character varying (20)	averagerating double precision	numvotes integer
1	tt9999999	5	120001

Fig. 21. Data before Transaction

Before updating the data, data confirmation is needed, as shown in Fig. 25. In this example, the entire transaction fails because the first update violates the constraint that $averageRating$ must be between 0 and 10. The transaction rolls back completely, preventing any partial updates and preserving database consistency. This behavior occurs regardless of operation order—even if the invalid update appeared later in the sequence, the transaction would still abort to maintain data integrity.

Data Output | Messages | Notifications

```
ERROR:  Invalid rating 89. Must be between 0 and 10.
CONTEXT:  PL/pgSQL function log_invalid_rating() line 3 at RAISE

SQL state: P0001
```

Fig. 22. Transaction Fail

To enforce these constraints and track errors, we implemented validation triggers. The validate_rating_trigger automatically checks whether $averageRating$ falls within the valid range before allowing any insert or update operation on the ratings table. If validation fails, the trigger logs detailed error messages (such as "Invalid rating 89: must be between 0 and 10") to assist with debugging, as is shown in Fig. 22.

Error logging serves an important purpose because database clients often provide unclear or transient error messages. By maintaining persistent logs, administrators can review and locate issues via more details of failures. This approach ensures data quality while providing transparency for troubleshooting.

When the correct operations are performed, such as setting $numVotes$ to 12000 and $averageRating$ to 8.1 and 7.9, the transaction succeeds. When modifying the same data at the same time, the database saves the latest modification, as shown in Fig. 23.

	tconst [PK] character varying (20)	averagerating double precision	numvotes integer
1	tt9999999	7.9	12000

Fig. 23. Transaction Success

VIII. Indexing Query Execution Analysis

The original database contains some inefficient designs that cause certain operations to run slowly. Three examples of such problematic queries are listed below.

Problematic Query 1

```
SELECT * FROM episode where parenttconst
    = 'tt0165593'
```

Problematic Query 2

```
select * from people p where p.primaryname
    = 'Jiabao'
```

Problematic Query 3

```
select tm.primarytitle, p2.primaryname
    from people p2
natural join profession_people pp
natural join profession p
natural join crew c
natural join tv_movie tm
where profession_name = 'director' and p2.
    primaryname = 'Jiabao'
```

These three queries take noticeably longer to execute, sometimes a few seconds or more. Using `EXPLAIN ANALYZE` to print the detailed execution plans reveals that their costs are relatively high.

	QUERY PLAN
1	Gather (cost=1000.00..13801.27 rows=73 width=28) (actual time=0.242..52.530 rows=121 loops=1)
2	Workers Planned: 1
3	Workers Launched: 1
4	Buffers: shared hit=6313
5	-> Parallel Seq Scan on episode (cost=0.00..12793.97 rows=43 width=28) (actual time=21.716..45.564 rows=60 loops=2)
6	Filter: ((parenttconst)::text = 'tt0165593'::text)
7	Rows Removed by Filter: 440646
8	Buffers: shared hit=6313
9	Planning:
10	Buffers: shared hit=6
11	Planning Time: 0.116 ms
12	Execution Time: 52.567 ms

Fig. 24. Detailed Execution Plan for Problematic Query 1

```
QUERY PLAN
|
------------------------------------------------------------------------------------------
------------+
Gather  (cost=1000.00..10198.98 rows=1 width=32) (actual time=33.459..37.610 rows=0 loops=1)
|
  Workers Planned: 1
|
  Workers Launched: 1
|
  Buffers: shared hit=4523
|
  ->  Parallel Seq Scan on people p  (cost=0.00..9198.88 rows=1 width=32) (actual time=30.819..30.819
rows=0 loops=2)|
        Filter: ((primaryname)::text = 'Jiabao'::text)
|
        Rows Removed by Filter: 317960
|
        Buffers: shared hit=4523
|
Planning:
|
  Buffers: shared hit=6 dirtied=1
|
Planning Time: 0.111 ms
|
Execution Time: 37.638 ms
|
```

Fig. 25. Detailed Execution Plan for Problematic Query 2

```
QUERY PLAN
------------------------------------------------------------------------------------------
Nested Loop  (cost=1.83..5176.73 rows=1 width=33) (actual time=616.616..616.619 rows=0 loops=1)
  Buffers: shared hit=433105
  ->  Nested Loop  (cost=1.42..5176.28 rows=1 width=24) (actual time=616.616..616.618 rows=0 loops=1)
        Join Filter: ((p2.nconst)::text = (c.nconst)::text)
        Buffers: shared hit=433105
        ->  Nested Loop  (cost=1.00..3781.47 rows=1 width=34) (actual time=616.615..616.617 rows=0 loops=1)
              Buffers: shared hit=433105
              ->  Nested Loop  (cost=0.57..847.95 rows=6496 width=10) (actual time=0.024..28.402 rows=108171 loops=1)
                    Buffers: shared hit=421
                    ->  Index Scan using profession_profession_name_key on profession p  (cost=0.14..2.36 rows=1 width=4) (actual time=0.009..0.011 rows=1 loops=1)
                          Index Cond: ((profession_name)::text = 'director'::text)
                          Buffers: shared hit=2
                    ->  Index Only Scan using profession_people_pkey on profession_people pp  (cost=0.43..579.86 rows=26573 width=14) (actual time=0.012..16.128 rows=108171 loops=1)
                          Index Cond: (profession_id = p.profession_id)
                          Heap Fetches: 0
                          Buffers: shared hit=419
              ->  Index Scan using people_pkey on people p2  (cost=0.42..0.45 rows=1 width=24) (actual time=0.005..0.005 rows=0 loops=108171)
                    Index Cond: ((nconst)::text = (pp.nconst)::text)
                    Filter: ((primaryname)::text = 'Jiabao'::text)
                    Rows Removed by Filter: 1
                    Buffers: shared hit=432684
        ->  Index Only Scan using crew_pkey on crew c  (cost=0.42..1394.79 rows=2 width=20) (never executed)
              Index Cond: (nconst = (pp.nconst)::text)
              Heap Fetches: 0
  ->  Index Scan using tv_movie_pkey on tv_movie tm  (cost=0.42..0.45 rows=1 width=29) (never executed)
        Index Cond: ((tconst)::text = (c.tconst)::text)
Planning:
  Buffers: shared hit=58
Planning Time: 1.217 ms
Execution Time: 616.673 ms
```

Fig. 26. Detailed Execution Plan for Problematic Query 3

By analyzing the detailed execution plans, it is evident that the cost of executing each query can exceed 10,000 in the worst cases. Careful examination of the three queries reveals that the main issue is the lack of indexing on several high-cost columns. For instance, in Problematic Query 1, a full table scan is performed on `Episode` because `parenttconst` is not indexed. Since there are tens of thousands of distinct `tconst` values, this leads to long execution times. Similarly, in Problematic Query 2, a sequential scan occurs because `primaryname` is not indexed. In Problematic Query 3, multiple `JOIN`s between tables without appropriate indexing further worsen the performance.

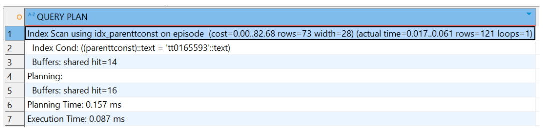
In order to address the performance issues, proper indexing should be implemented. As mentioned above, the high-cost columns `parenttconst` in `Episode` and `primaryname` in `People` should be indexed. Since `parenttconst` serves as a unique identifier for each TV series record and is primarily used for equality searches, a hash index is applied because it is optimized specifically for equality lookups. For `primaryname`, although the example query involves an equality search, names are likely to be used in other operations such as ordering or range queries. Therefore, a B-tree index is used instead of a hash index.

It is also noted that other indexing strategies were attempted to improve the performance of problematic query 3, such as adding an index on `profession name`. However, the effect was not significant, probably because `profession name` has only around 30 distinct values, making an index no more efficient than a sequential scan.

The SQL commands for creating the indexes are shown below.

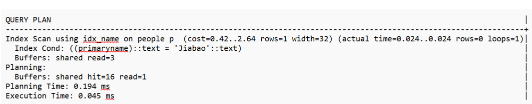
```
CREATE INDEX idx_parenttconst ON episode
    USING hash (parenttconst);
CREATE INDEX idx_name ON people(
    primaryname);
```

The detailed execution plan after applying index for the 3 queries are shown below.



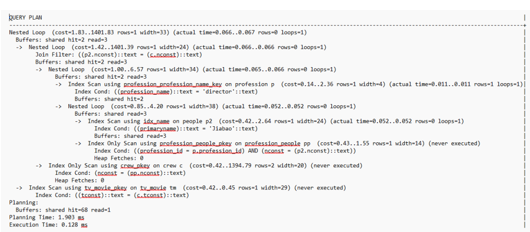
	QUERY PLAN
1	Index Scan using idx_parenttconst on episode (cost=0.00..82.68 rows=73 width=28) (actual time=0.017..0.061 rows=121 loops=1)
2	Index Cond: ((parenttconst)::text = 'tt0165593'::text)
3	Buffers: shared hit=14
4	Planning:
5	Buffers: shared hit=16
6	Planning Time: 0.157 ms
7	Execution Time: 0.087 ms

Fig. 27. Detailed Execution Plan for Query 1 with Index



```
QUERY PLAN
-------------------------------------------------------------------------------
Index Scan using idx_name on people p  (cost=0.42..2.64 rows=1 width=32) (actual time=0.024..0.024 rows=0 loops=1)
  Index Cond: ((primaryname)::text = 'Jiabao'::text)
  Buffers: shared read=3
Planning:
  Buffers: shared hit=16 read=1
Planning Time: 0.194 ms
Execution Time: 0.045 ms
```

Fig. 28. Detailed Execution Plan for Query 2 with Index



```
QUERY PLAN
-------------------------------------------------------------------------------
Nested Loop  (cost=1.83..1401.83 rows=1 width=33) (actual time=0.066..0.067 rows=0 loops=1)
  Buffers: shared hit=2 read=3
  ->  Nested Loop  (cost=1.42..1401.39 rows=1 width=24) (actual time=0.066..0.066 rows=0 loops=1)
        Join Filter: ((p2.nconst)::text = (c.nconst)::text)
        Buffers: shared hit=2 read=3
        ->  Nested Loop  (cost=1.00..6.57 rows=1 width=34) (actual time=0.065..0.066 rows=0 loops=1)
              Buffers: shared hit=2 read=3
              ->  Index Scan using profession_profession_name_key on profession p  (cost=0.14..2.36 rows=1 width=4) (actual time=0.011..0.011 rows=1 loops=1)
                    Index Cond: ((profession_name)::text = 'director'::text)
                    Buffers: shared hit=2
              ->  Nested Loop  (cost=0.85..4.20 rows=1 width=38) (actual time=0.052..0.052 rows=0 loops=1)
                    Buffers: shared read=3
                    ->  Index Scan using idx_name on people p2  (cost=0.42..2.64 rows=1 width=24) (actual time=0.052..0.052 rows=0 loops=1)
                          Index Cond: ((primaryname)::text = 'Jiabao'::text)
                          Buffers: shared read=3
                    ->  Index Only Scan using profession_people_pkey on profession_people pp  (cost=0.43..1.55 rows=1 width=14) (never executed)
                          Index Cond: ((profession_id = p.profession_id) AND (nconst = (p2.nconst)::text))
                          Heap Fetches: 0
        ->  Index Only Scan using crew_pkey on crew c  (cost=0.42..1394.79 rows=2 width=20) (never executed)
              Index Cond: (nconst = (pp.nconst)::text)
              Heap Fetches: 0
  ->  Index Scan using tv_movie_pkey on tv_movie tm  (cost=0.42..0.45 rows=1 width=29) (never executed)
        Index Cond: ((tconst)::text = (c.tconst)::text)
Planning:
  Buffers: shared hit=68 read=1
Planning Time: 1.903 ms
Execution Time: 0.128 ms
```

Fig. 29. Detailed Execution Plan for Query 3 with Index

The execution plans demonstrate that indexing significantly improves performance. For example, the cost of executing Query 1 was reduced from over 10,000 to only 83 in the worst case. Execution costs for the other two queries also saw similar dramatic improvements, confirming that the indexing strategy effectively resolved the performance issues.

IX. Visualization Power BI - Gary

A Power BI report is created to display some insights obtained from the database. The report consists of three pages in total.Page 1 provides an overview of the movie/TV database.Pages 2 and 3 present insights into TV series and movies, respectively.The screenshots of each page were presented below. However, it includes dynamic slicer to allow the user browser the visualization freely. The report can be viewed by this link Power BI Dashboard

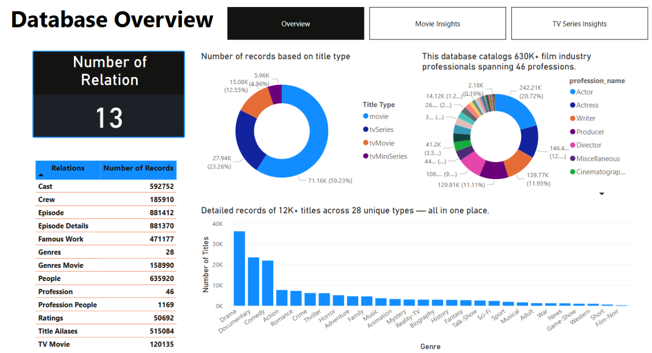


Fig. 30. Page 1 – Database Overview

Fig. 31. Page 2 – Movie Insights

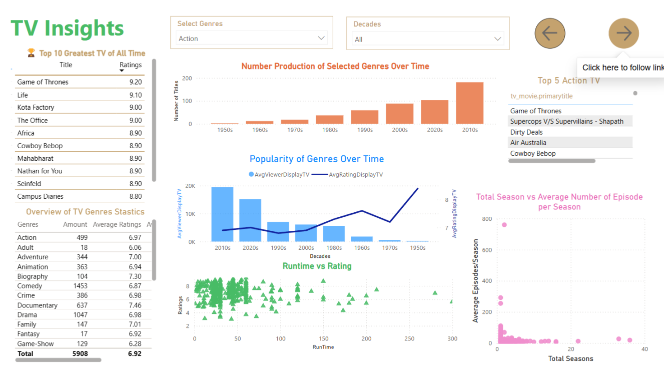


Fig. 32. Page 3 – TV Insights

A. Queries

Except for the original relations, several SQL queries were executed when importing the database into Power BI. The details of all queries and their outputs are presented in Section

VI. It is also noted that some visualization are generated using original relations or with the support of built-in Power BI features such as Power Query Editor and Measures for the convenience of design.

Visualizations	Query
Top 10 Movie/TV of ALL Time	Query H
Statistics of Film/TV By Genre	Query K
Number of Production of Genres Over Time	Query L
Popularity of Genres Over Time	Query L
Top Film/TV Based on Title Type and Genre	Query N
Professionals Specializing in Specific Genres	Query J

TABLE I
YOUR CAPTION HERE

B. Insights

Number of Relations: show the total number of relations this database contains.
Number of Records: This shows the number of records for each relations. It provides a straightforward information about the distribution across genres.
Number of Records Based on Title Type: This shows the distribution by title type. From the graph, it can be seen that movies account for about 70% of the total records. Adding more TV titles could help balance the database and improve diversity.
Film Profession Distribution: This shows the distribution of people records by job category.
Number of Works by Genre: This shows the number of records for each genre, combining both TV and movies. It provides straightforward information about the distribution across genres.
Top 10 Greatest Movies/TV Shows of All Time: This table shows the top 10 highest-rated movies or TV series according to IMDb users. It is a classic list that is of great interest to cinephiles.
Overview of Film/TV Genre Statistics: This table shows the number of records for each genre, specific to films or TV series, along with their average ratings and average number of viewers. From the table, we can gain a good understanding of the data distribution.
Number of Productions of Selected Genre Over Time: Based on the selection, we can observe how the number of works in a particular genre changes over time. This reflects the rise and fall of different types of film works and shows the shifts in public interest. For example, Western films were very popular from the 1930s to the 1950s, but their number has significantly declined over time, illustrating the fading of this once-popular genre.
Popularity of Genres Over Time: Based on the selection, we can observe how modern audiences perceive works from different time periods. Generally, viewership for classic TV shows and movies is lower compared to modern productions, but their ratings tend to be higher, indicating that older works still have a fan base. Filmmakers can also identify which genres are currently more popular, providing valuable insights for production and marketing decisions.
Runtime and Ratings: This visualization shows the distribution of runtime and ratings for each movie within a selected genre. It can be observed that different genres have different average ratings and runtime. For example, documentaries are more likely to have runtime exceeding 150 minutes, which is much less common in other genres.
Top Talent in Current Genre: This table displays individuals in the selected job category who are particularly suitable for the genre, along with their famous works. Suitability is determined based on the average ratings of the titles they participated in within the selected genre. This feature can help production companies quickly identify and select desired candidates for specific jobs or roles.
Top 5 Movies/TV: This table shows the most popular five movies or TV series in the selected genre. It can help production companies find good references when starting a new project.
Percentage of Adult Works: This pie chart shows the proportion of restricted-rated works within a selected genre. It provides interesting insights into the maturity level of different genres. For example, there are more crime movies rated as restricted compared to animation. **Total Seasons vs Average Number of Episodes**: Besides the visualizations mentioned above, TV insights also include this one. It reveals the relationship between the number of seasons and the average number of episodes per season. From the visualization, we can observe that most TV series have around 10 seasons, but shows with more seasons tend to have fewer episodes per season which matches the current trends in TV series.

X. Demo and Presentation - Jiabao

XI. Readme- Together

References

[1] IMDb, "IMDb Non-Commercial Datasets" [Online]. Available: https://developer.imdb.com/non-commercial-datasets/. [Accessed: Mar. 16, 2025].
[2] H. García-Molina, J. D. Ullman, and J. Widom, "Database Systems: The Complete Book", 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2008.
[3] A. Silberschatz, H. F. Korth, and S. Sudarshan, "Database System Concepts", 7th ed. New York, NY, USA: McGraw-Hill, 2019.
[4] R. E. Silverman, "Bad Decisions in Film: Missteps, Missed Opportunities, and Miscasting", London: Routledge, 2013.