DMQL Milestone-2

Project report on Movie Database

Academic Integrity: “We have read and understood the course academic integrity policy in the syllabus of the class.”

Project Details

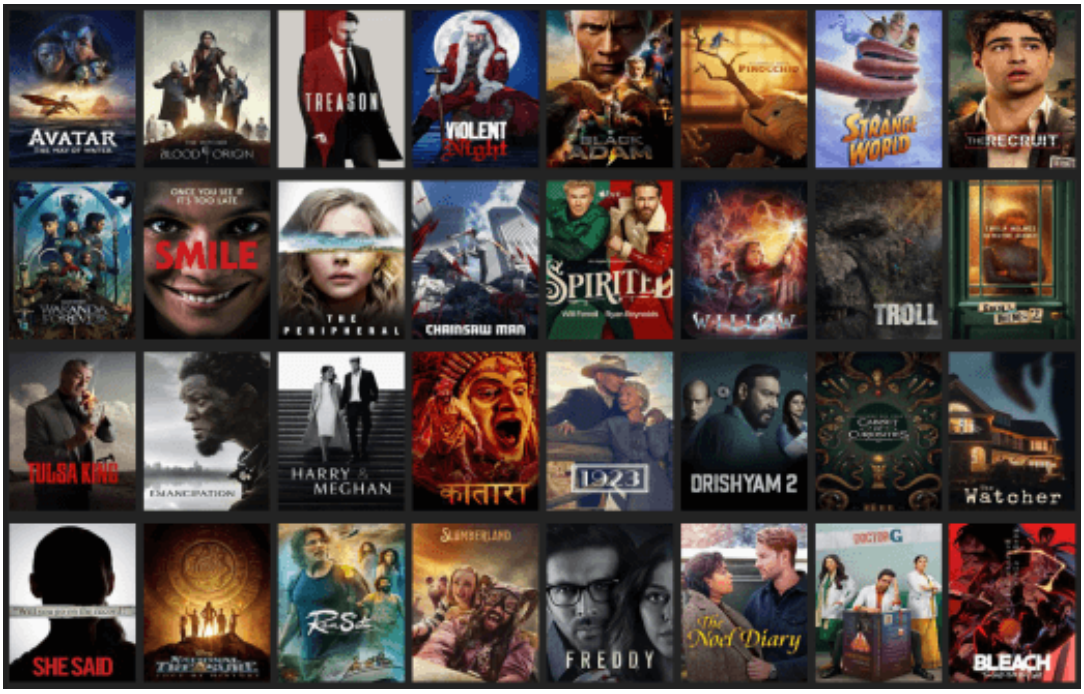
Project Name: The Movie Database

Assigned TA: Jingchen

Team Number: 4

Team Members:

1. Yasaschandra Naga Sai Kasaraneni (UBIT: **yasascha**)
2. Danussh Garlapati (UBIT: **danusshg**)
3. Harshitha Siddaramaiah (UBIT: **hsiddara**)



Problem Statement: Our project aims at designing and implementing a relational database management system (RDBMS) that stores and manages information about movies/TV shows and their remakes, including their titles, genres, release dates, production companies, budgets, and box office revenues. The database also stores information about the people involved in making the movies, such as directors, actors and their awards, producers and their studios, and their respective roles in each movie.

The RDBMS should allow for efficient querying of the database to answer questions such as:

- The RDBMS should allow for efficient querying of the database to answer questions such as:
- Which movies have been released in a given year or range of years?
- Which movies were directed by a particular director or produced by a particular production company?
- Which actors have appeared in the most movies, and which genres are they most commonly associated with?

The ultimate goal is to create a comprehensive and reliable source of information that can support decision-making in the movie industry, such as selecting actors and directors, planning production schedules, and marketing campaigns.

The system allows movie producers, directors, and other stakeholders in the movie industry to efficiently store, retrieve, and manage information about movies, actors, directors, production companies, genres, and other relevant entities. The database designed should be able to handle a large amount of data, while ensuring data integrity, consistency, and security. We are also aiming to provide an intuitive user interface that is easy to use and visually appealing to movie enthusiasts.

In the future, this storage system can additionally be extended to provide personalized movie recommendations based on a user's viewing history and preferences. Overall, the goal is to create a comprehensive and user-friendly RDBMS that provides valuable insights into the movie industry and enhances the movie-watching experience for users.

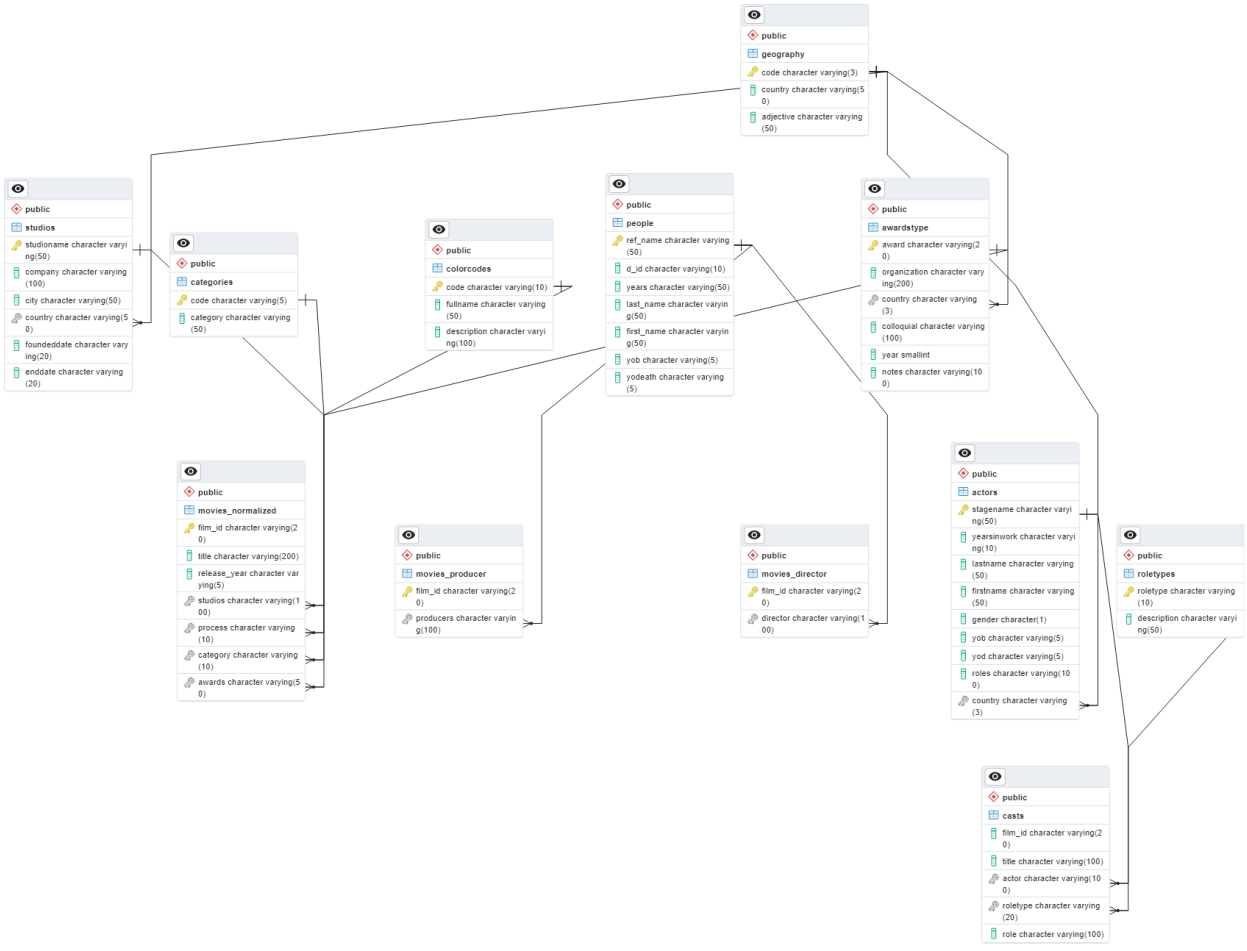
Target User: The target users of our system can include a wide range of individuals and organizations in the movie industry, as well as researchers and enthusiasts interested in studying or analyzing movie-related data.

Some potential target users include:

1. **Movie production companies**, who can use the database to track and analyze data on their own movies, as well as competitors' movies, and make informed decisions about production, marketing, and distribution. For ex: To decide which movies to invest in and how much to budget for production and marketing.

2. **Film distributors and exhibitors**, who can use the database to identify popular movies and trends, and adjust their offerings accordingly.
3. **Movie critics and reviewers**, who can use the database to research and analyze movies and write informed reviews.
4. **Researchers and academics**, who can use the database to study trends in the movie industry, analyze movie-related data, and develop new insights and theories.
5. **General movie enthusiasts**, who can use the database to discover new movies, explore movie-related data, and engage with other movie fans.

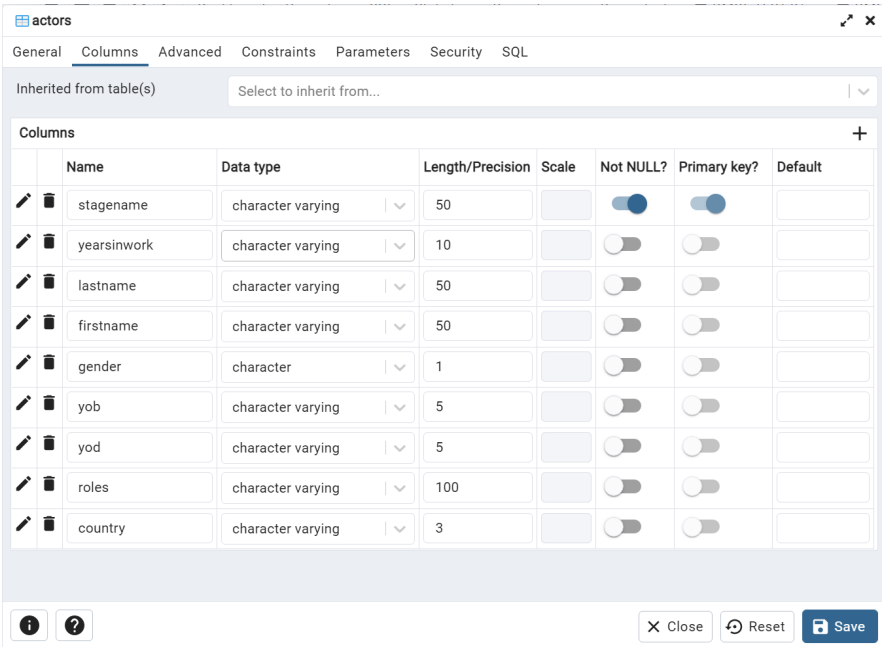
ER Diagram:



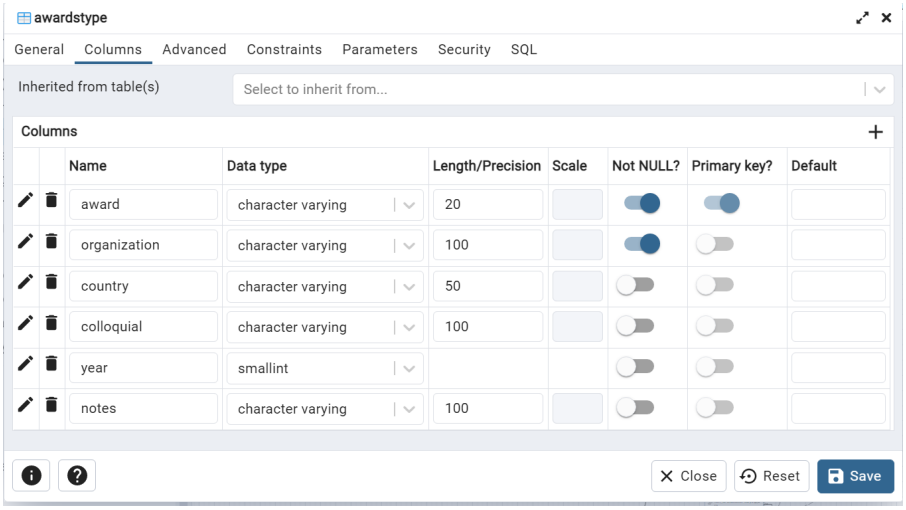
Database and Tables:

Database Schemas:

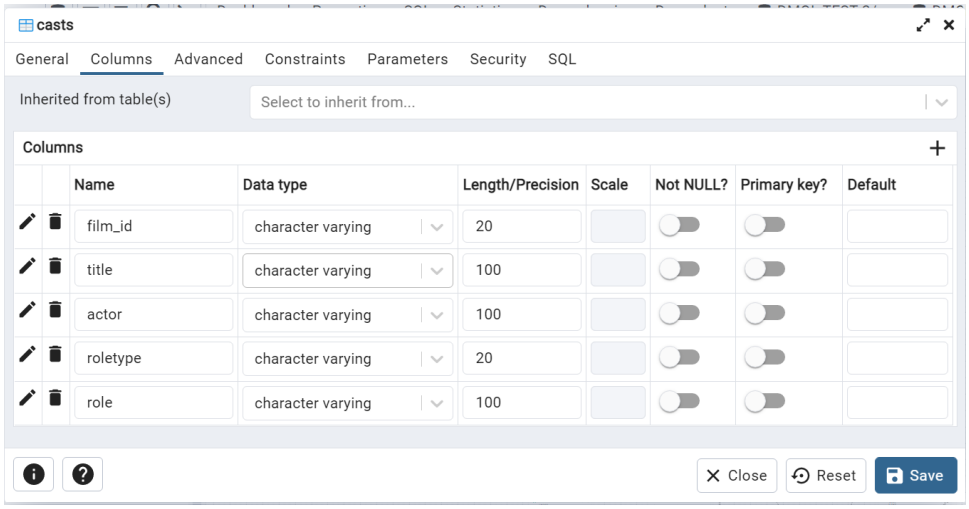
1. Actors table:



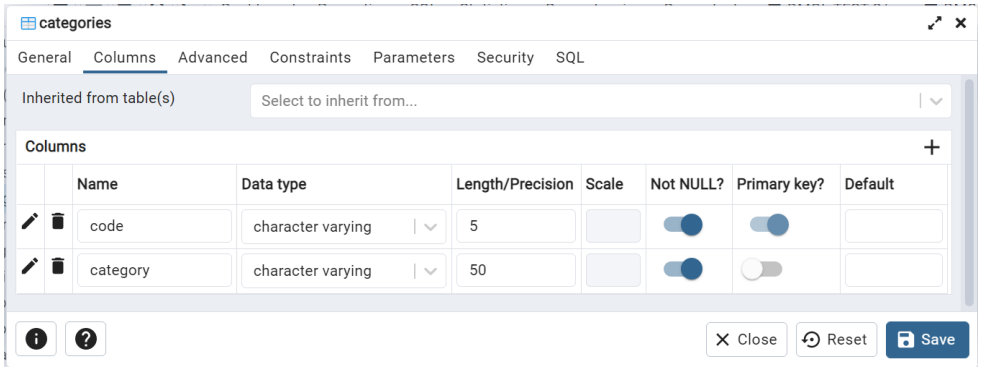
2. Awardstype table:



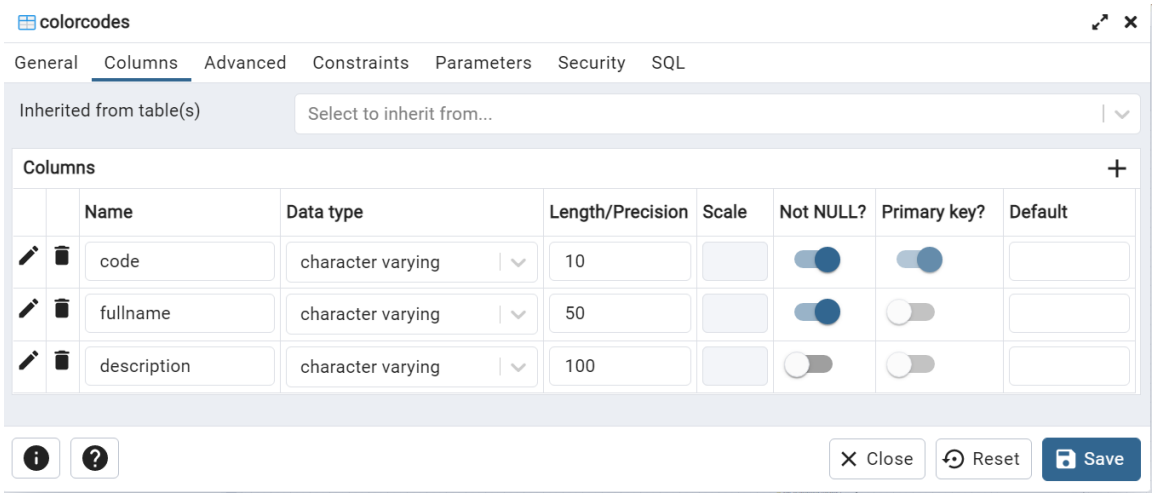
3. Casts table:



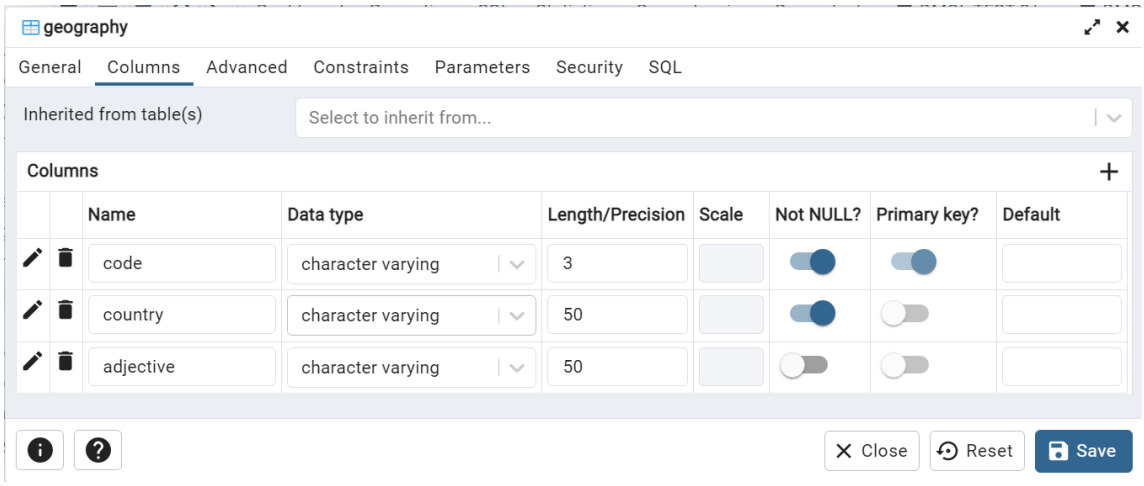
4. Categories table:



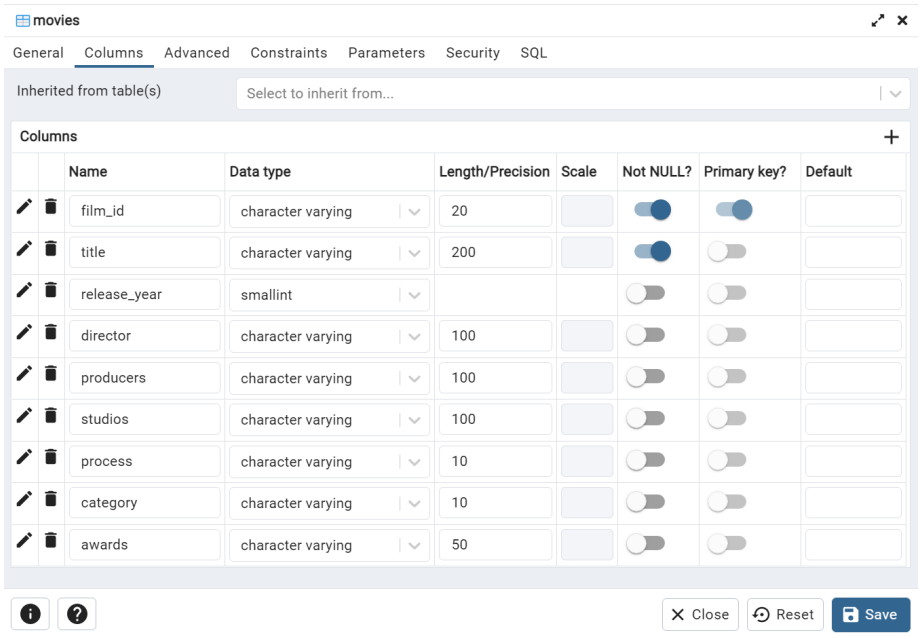
5. Colorcodes table:



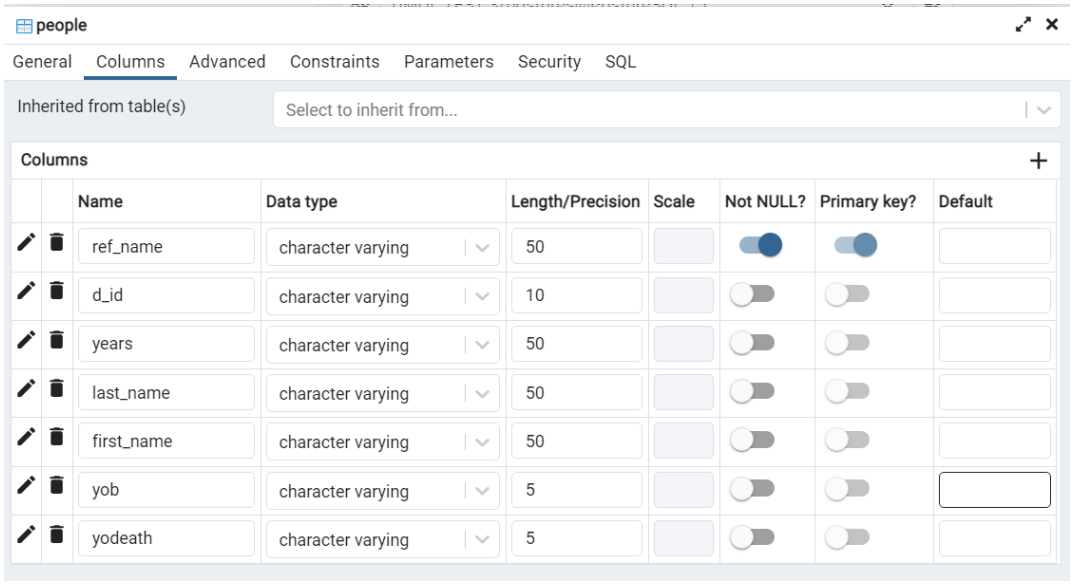
6. Geography table:



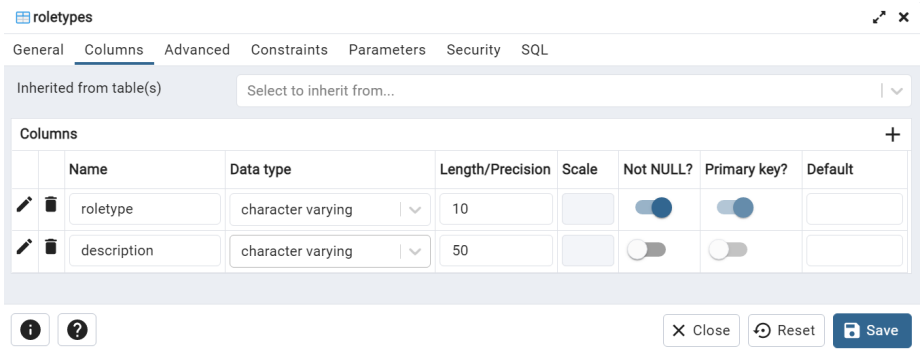
7. Movies table:



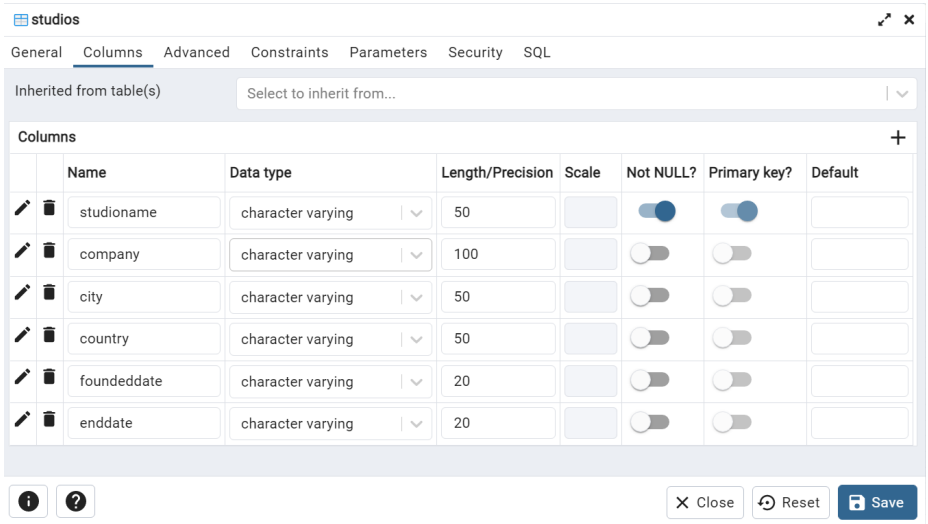
8. People table:



9. Roletypes table:



10. Studios table:



Relationship Between tables:

Attributes:

There are 10 Tables in the database namely:

1. Actor Table:

Attribute	Attribute Description	Data type	Constraints
stagename	Stagename of the actor.	varchar(50)	PRIMARY KEY

yearsinwork	Years in total work	varchar(10)	DEFAULT: NA
lastname	Original last name	varchar(50)	DEFAULT: NA
firstname	Original first name	varchar(50)	DEFAULT: NA
gender	coded as M,F, and X for unknown	character(1)	DEFAULT: NA
yob	Date of Birth.	varchar(5)	DEFAULT: NA
yod	Date of Death	varchar(5)	DEFAULT: NA
roles	Types of roles played by the actor	varchar(100)	DEFAULT: NA
country	Country of origin using COUNTRY-CODES	varchar(3)	FOREIGN KEY

2. Awardstype table

Attribute	Attribute Description	Data type	Constraints
award	Award	varchar(20)	PRIMARY KEY
organization	Organization giving the award	varchar(100)	NOT NULL

country	Country award was originated	varchar(50)	
colloquial	Simpler description of the award	varchar(100)	
year	Year in which award is introduced	smallint	
notes	Notes about award	varchar(100)	

3. Casts table

Attribute	Attribute Description	Data type	Constraints
film_id	Id of the movie	varchar(20)	FOREIGN KEY
title	Title of the movie	varchar(100)	
actor	Actor in the movie	varchar(100)	FOREIGN KEY
roletype	Type of role the person played	varchar(20)	FOREIGN KEY
role	Role of the person	varchar(100)	

4. Categories table

Attribute	Attribute Description	Data type	Constraints
code	Code assigned to the genre	varchar(5)	PRIMARY KEY
category	Genre name	varchar(50)	NOT NULL

5. Colorcodes table

Attribute	Attribute Description	Data type	Constraints
code	Code assigned for the process	varchar(10)	PRIMARY KEY
fullname	Name of the process	varchar(50)	NOT NULL
description	Description of the process	varchar(100)	

6. Geography table

Attribute	Attribute Description	Data type	Constraints
code	Code assigned to the country	varchar(3)	PRIMARY KEY
country	Name of the country	varchar(50)	NOT NULL
adjective	How a person is called if he belongs to that country	varchar(50)	

7. Movies table

Attribute	Attribute Description	Data type	Constraints
film_id	Id assigned to the Movie	varchar(20)	PRIMARY KEY
title	Title of the movie	varchar(200)	NOT NULL
release_year	Year in which the movie is released	smallint	
director	Director of the movie	varchar(100)	FOREIGN KEY

producers	Producer of the movie	varchar(100)	FOREIGN KEY
studios	Studio that produced the movie	varchar(100)	FOREIGN KEY
process	Mode in which movie is made	varchar(100)	FOREIGN KEY
category	The movie genre	varchar(100)	FOREIGN KEY
awards	Awards awared to the movie	varchar(100)	FOREIGN KEY

8. People table

Attribute	Attribute Description	Data type	Constraints
ref_name	Reference name of the person	varchar(50)	PRIMARY KEY
d_id	Id if the person is Director	varchar(10)	
years	Years the person worked in Industry	varchar(50)	
last_name	Last name of the	varchar(50)	

	person		
first_name	First name of the person	varchar(50)	
yob	Year in which the person is born	varchar(5)	
yodeath	Year in which the person died	varchar(5)	

9. Roletype table

Attribute	Attribute Description	Data type	Constraints
roletype	Characters representing the role cast played for the movie	varchar(10)	PRIMARY KEY
description	Role Description	varchar(50)	

10. Studios table

Attribute	Attribute Description	Data type	Constraints
studioname	Name of the studio	varchar(50)	PRIMARY KEY
company	Company maintaining the studio	varchar(100)	
city	City in which the city is located	varchar(50)	
country	Country where the studio is located	varchar(50)	
foundeddate	Date when the studio is established	varchar(20)	
enddate	Date when if the studio shut down	varchar(20)	

Slight Description of the Tables:

- Actor Tables contains the data regarding actor which is firstname, lastname, gender, yearofbirth, yearofdeath, roles, country of acting etc.
    - There is one record for each actor listed, but not all actors listed in CAST are documented. There are also break and header records for each letter of the alphabet.
    - Stagenm = Stagename of the actor. This is nearly the key of the table. When an actor has used multiple names the last one used is preferred. There are a few actors with identical names. Then the birthyear (dob) becomes important.
    - **Stagename** column is the **primarykey** of the table

- Awardstype table contains the award name, organisation, country, year etc.
    - Award column is the primary key of the table
- Casts Table contains film_id, title, actor, roletypes and role
    - The large CAST file is broken up into sections by initial letter(s) of the directors identifying code. In each section will be a number of directors, ordered by code. There is a distinct table for each director.
    - Doesnt contain the primary key for the table, acts as a helper table.
- Categories Table contains code and categories.
    - Code is the primary key for the table.
- Colorcodes Table contains code, fullname, and description.
    - Code column is the primary key of the table.
- Geography Table contains code, country, and adjective.
    - Code column is the primary for the table.
- Movies table contains film_id, title, release_year, director, producers, studios, process, category and awards.
    - There is a distinct table for each director (Hitchcock has multiple tables, one for early silent, one for British, one for American, and one for TV movies).The tables are broken up by year of first known film by the directors. There are some break and header records for each year.
    - Film_id column is the primary key of the table.
- People table contains ref_name, codes, d_ide, years, last_name, first_name, yob etc
    - Directors are the major subset of the general people.html table. Other entries are significant producers, writers, art directors and some authors. Being a director is indicated in the Pcode field, and has some effect on other fields. There are also break and header records for each letter of the alphabet
    - Ref_name is the primary key of the table.
- Roletype table: contains roletype and description columns.
    - Roletype column is the primary key of the table.
- Studios table: contains studioname, company, city, country, foundeddate and enddate.
    - Studioname column is the primary key of the table.

Primary and Foreign Key:

1. Actors Table:
    a. Primary Key - stagename
    b. Foreign Key - country which is REFERENCED from public.geography (code)
2. Awardstype table:
    a. Primary Key - award
    b. Foreign Key - No Foreign Key
3. Casts table:
    a. Primary Key - No Primary Key
    b. Foreign Key -
        i. Actor which is REFERENCED from public.actors (stagename)
        ii. Film_id which is REFERENCED from public.movies (film_id)
        iii. Roletype which is REFERENCED public.roletypes (roletype)
4. Categories table:
    a. Primary Key - code
    b. Foreign Key - No Foreign Key
5. Colorcodes table:
    a. Primary Key - code
    b. Foreign Key - No Foreign Key
6. Geography table:
    a. Primary Key - code
    b. Foreign Key - No Foreign Key
7. Movies table:
    a. Primary Key - film_id
    b. Foreign Key -
        i. Awards which is REFERENCED from public.awardstype (award)

ii. Category which is REFERENCED from public.categories (code)

iii. Director which is REFERENCED from public.people (ref_name)

iv. Process which is REFERENCED from public.colorcodes (code)

v. Studios which is REFERENCED from public.studios (studioname)

8. People table:
    a. Primary Key - ref_name
    b. Foreign Key - No Foreign Key

9. Roletype table:
    a. Primary Key - roletype
    b. Foreign Key - No Foreign Key

10. Studios table:
    a. Primary Key - studioname
    b. Foreign Key - No Foreign Key

Normalization

To determine whether each relation is in Boyce-Codd Normal Form (BCNF), we need to check if there are any functional dependencies (FDs) that violate BCNF. A relation is in BCNF if every non-trivial FD (i.e., an FD that cannot be inferred from the candidate keys) has a determinant that is a superkey. In other words, no non-trivial FD should have a determinant that is not a candidate key.

Let's start with the roletypes relation:

1. **roletypes**

    Functional Dependencies:

    roletype -> description

This relation has a single candidate key, which is roletype. The FD in this relation is in BCNF because the determinant (roletype) is a candidate key. Therefore, this relation is already in BCNF.

2. **colorcodes**

   Functional Dependencies:

   code -> fullname, description

This relation has a single candidate key, which is code. The FD in this relation is in BCNF because the determinant (code) is a candidate key. Therefore, this relation is already in BCNF.

3. **categories**

   Functional Dependencies:

   code -> category

This relation has a single candidate key, which is code. The FD in this relation is in BCNF because the determinant (code) is a candidate key. Therefore, this relation is already in BCNF.

4. **geography**

   Functional Dependencies:

   code -> country, adjective

This relation has a single candidate key, which is code. The FD in this relation is in BCNF because the determinant (code) is a candidate key. Therefore, this relation is already in BCNF.

5. **studios**

   Functional Dependencies:

   studioname -> company, city, country, foundeddate, enddate

This relation has a single candidate key, which is studioname. The FD in this relation is in BCNF because the determinant (studioname) is a candidate key. Therefore, this relation is already in BCNF.

6. **people**

Functional Dependencies:

ref_name -> d_id, years, last_name, first_name, yob, yodeath

This relation is already in BCNF since the primary key determines all the attributes in the relation.

7. **awardstype**

Functional Dependencies:

award -> organization, country, colloquial, year, notes

This relation has a single candidate key, which is ‘award’. The FD in this relation is in BCNF because the determinant (award) is a candidate key. Therefore, this relation is already in BCNF.

8. **actors**

Functional Dependencies:

stagename -> yearsinwork, lastname, firstname, gender, yob, yod, roles, country

This relation has a single candidate key, which is stagename. The FD in this relation is in BCNF because the determinant (stagename) is a candidate key. Therefore, this relation is already in BCNF.

9. **movies**

The movies table is not in BCNF because there are several partial functional dependencies on the candidate key film_id. Specifically, the attributes director, producers, and studios are all functionally dependent on film_id, which is the candidate key. To normalize this table, we need to split it into three tables: movies, directors and producers.

The new movies table has the following columns:

film_id (primary key)
title
release_year
process
category
awards

The new movies_directors table has the following columns:

film_id (foreign key)
ref_name (foreign key)

The new movies_producers table has the following columns:

film_id (foreign key)
ref_name (foreign key)

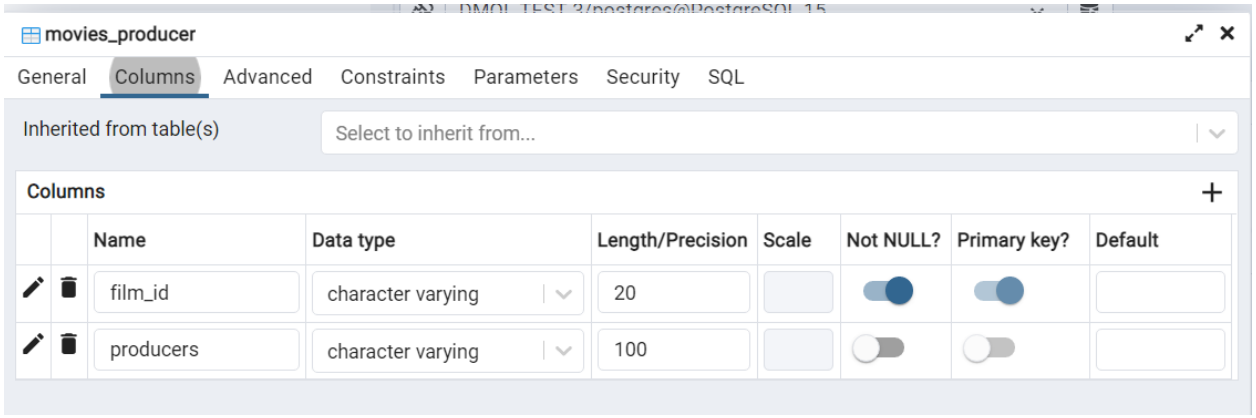
The new movies_studios table has the following columns:

film_id (foreign key)
studioname (foreign key)

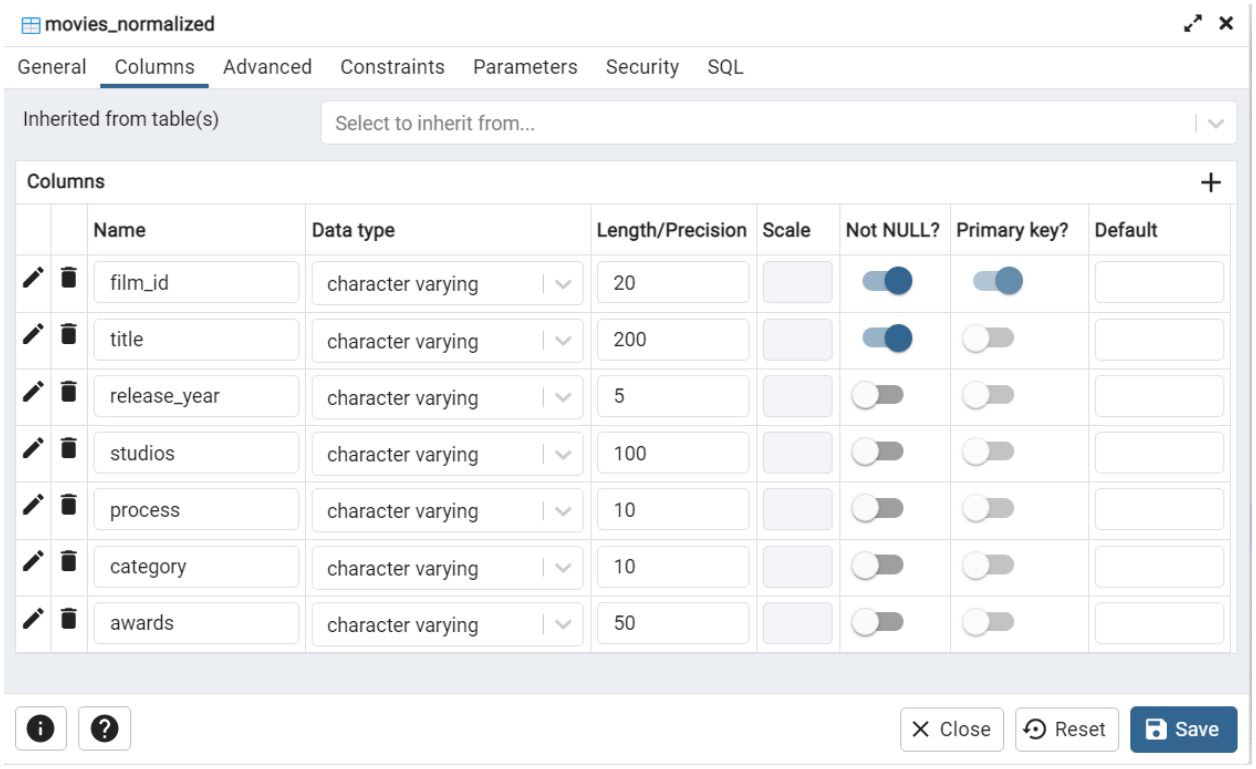
Functional Dependencies:

film_id -> title, release_year, process, category, awards
director -> ref_name
producers -> ref_name
studios -> studioname

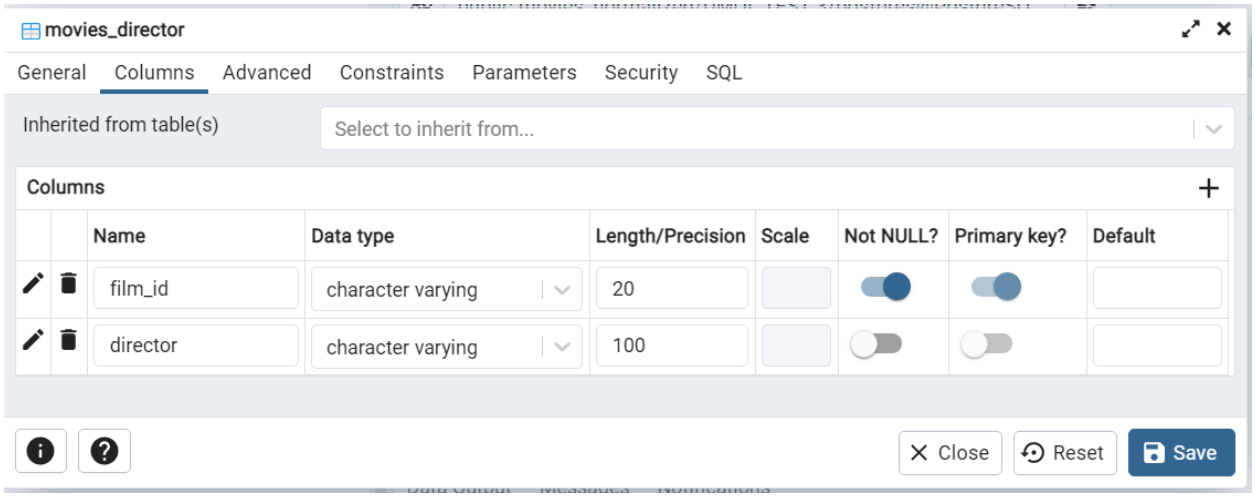
Movie_producer:



Movies_normalized:



Movies_director:



10. casts

Functional Dependencies: film_id, actor, roletype → title, role

This table is already in BCNF as there are no partial dependencies or transitive dependencies. All attributes are functionally dependent on the candidate key (film_id, actor, roletype).

No non-trivial functional dependencies.

Sample Database:

Create Script:

1. **Actors Table:**

```
CREATE TABLE actors
(
    stagename varchar(50) NOT NULL,
    yearsinwork varchar(10),
    lastname varchar(50),
    firstname varchar(50),
    gender character(1),
    yob varchar(5),
    yod varchar(5),
    roles varchar(100),
    country varchar(3),
    CONSTRAINT actors_pkey PRIMARY KEY (stagename),
    CONSTRAINT actors_country_fkey FOREIGN KEY (country)
        REFERENCES public.geography (code) MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE CASCADE
);
```

2. **Awardstype Table:**

```
CREATE TABLE awardstype
(
    award varchar(20) NOT NULL,
    organization varchar(100) NOT NULL,
    country varchar(50),
    colloquial varchar(100),
    year smallint,
    notes varchar(100),
    CONSTRAINT awardstype_pkey PRIMARY KEY (award)
);
```

3. **Casts Table:**

```
CREATE TABLE casts
(
    film_id varchar(20),
    title varchar(100),
    actor varchar(100),
    roletype varchar(20),
    role varchar(100),
    CONSTRAINT casts_actor_fkey FOREIGN KEY (actor)
        REFERENCES public.actors (stagename) MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE CASCADE,
    CONSTRAINT casts_film_id_fkey FOREIGN KEY (film_id)
        REFERENCES public.movies (film_id) MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE CASCADE,
    CONSTRAINT casts_roletype_fkey FOREIGN KEY (roletype)
        REFERENCES public.roletypes (roletype) MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE CASCADE
);
```

4. **Categories Table:**

```
CREATE TABLE categories
(
    code varchar(5) NOT NULL,
    category varchar(50) NOT NULL,
    CONSTRAINT categories_pkey PRIMARY KEY (code)
);
```

5. **Colorcodes Table:**

```
CREATE TABLE colorcodes
(
    code varchar(10) NOT NULL,
    fullname varchar(50) NOT NULL,
    description varchar(100),
    CONSTRAINT colorcodes_pkey PRIMARY KEY (code)
);
```

6. **Geography Table:**

```
CREATE TABLE geography
(
    code varchar(3) NOT NULL,
    country varchar(50) NOT NULL,
    adjective varchar(50),
    CONSTRAINT geography_pkey PRIMARY KEY (code)
);
```

7. **Movies Table:**

Query | Query History

```
CREATE TABLE IF NOT EXISTS public.movies
(film_id character varying(20) COLLATE pg_catalog."default" NOT NULL,
    title character varying(200) COLLATE pg_catalog."default" NOT NULL,
    release_year character varying(5) COLLATE pg_catalog."default",
    director character varying(100) COLLATE pg_catalog."default",
    producers character varying(100) COLLATE pg_catalog."default",
    studios character varying(100) COLLATE pg_catalog."default",
    process character varying(10) COLLATE pg_catalog."default",
    category character varying(10) COLLATE pg_catalog."default",
    awards character varying(50) COLLATE pg_catalog."default",
    CONSTRAINT movies_pkey PRIMARY KEY (film_id),
    CONSTRAINT movies_awards_fkey FOREIGN KEY (awards)
        REFERENCES public.awardstype (award) MATCH SIMPLE
        ON UPDATE CASCADE ON DELETE CASCADE,
    CONSTRAINT movies_category_fkey FOREIGN KEY (category)
        REFERENCES public.categories (code) MATCH SIMPLE
        ON UPDATE CASCADE ON DELETE CASCADE,
    CONSTRAINT movies_director_fkey FOREIGN KEY (director)
        REFERENCES public.people (ref_name) MATCH SIMPLE
        ON UPDATE CASCADE ON DELETE CASCADE,
    CONSTRAINT movies_process_fkey FOREIGN KEY (process)
        REFERENCES public.colorcodes (code) MATCH SIMPLE
        ON UPDATE CASCADE ON DELETE CASCADE,
    CONSTRAINT movies_producers_fkey FOREIGN KEY (producers)
        REFERENCES public.people (ref_name) MATCH SIMPLE
        ON UPDATE CASCADE ON DELETE CASCADE,
    CONSTRAINT movies_studios_fkey FOREIGN KEY (studios)
        REFERENCES public.studios (studioname) MATCH SIMPLE
        ON UPDATE CASCADE ON DELETE CASCADE
)
```

8. **People Table:**

Query | Query History

```
CREATE TABLE IF NOT EXISTS public.people
(
    ref_name character varying(50) COLLATE pg_catalog."default" NOT NULL,
    d_id character varying(10) COLLATE pg_catalog."default",
    years character varying(50) COLLATE pg_catalog."default",
    last_name character varying(50) COLLATE pg_catalog."default",
    first_name character varying(50) COLLATE pg_catalog."default",
    yob character varying(5) COLLATE pg_catalog."default",
    yodeath character varying(5) COLLATE pg_catalog."default",
    CONSTRAINT people_pkey PRIMARY KEY (ref_name)
)
```

9. **Roletype Table:**

```
CREATE TABLE roletypes
(
    roletype varchar(10) NOT NULL,
    description varchar(50),
    CONSTRAINT roletypes_pkey PRIMARY KEY (roletype)
);
```

10. **Studios Table:**

```
CREATE TABLE studios
(
    studioname varchar(50) NOT NULL,
    company varchar(100),
    city varchar(50),
    country varchar(50),
    foundeddate varchar(20),
    enddate varchar(20),
    CONSTRAINT studios_pkey PRIMARY KEY (studioname)
);
```

Insert Script:

1. **Actor Table:**

```
INSERT INTO actors (stagename, yearsinwork, lastname, firstname, gender, yob, yod, roles, country)
VALUES ('Willie Aames',NULL,'Aames',    'William',  'M',    '1960', '199x', NULL,'Am'),
       ('Bud Abbott',   '1939-1956',    'Abbott',   'William',  'M',    '1895', '1974', 'straight, comedian','Am'),
       ('Diahnne Abbott','1976-1982',NULL,NULL,'F',NULL,'199x','sexy','Am'),
       ('Leslie Howard',    '1930-1943',    'Stainer',  'Leslie'    ,'M'    ,1890,  1943,   'romantic intellectual',    'Hu'),
       ('George Abbott',    '1928-1958',    'Abbott',   'George',   'M',    '1887', '199x', 'playwright, producer','Am'),
       ('Wendy Hiller',     '1937-1982',    'Hiller',   'Wendy',    'F',    1912,   '199x', 'distinguished, inimitable voice','Br'),
       ('Marie Lohr',   '1932-1956',    'Lohr', 'Marie',    'F',    1890,   1975,   'distinguished',    'Au')
       ('Wilfrid Lawson'    ,'1936-1966',   'Worsnop'   ,'Wilfrid'  ,'M'    ,1900,  1966,   'character' ,'Br');
```

2. **Awardtypes table:**

```
INSERT INTO awardstype (award, organization, country, colloquial, year, notes)
VALUES ('AA',   'Hollywood Academy of Motion Picture Arts and Sciences',    'USA',  'Oscar', NULL,NULL),
       ('AAN', 'Hollywood Academy of Motion Picture Arts and Sciences',    'USA',  'Oscar nomination',NULL,NULL),
       ('AFI Lifetime',    'American Film Institute',  'USA',  'annual awards since 1973', 1973,NULL),
       ('AFI77',   'American Film Institute',  'USA',  'Best movies poll', 1977,NULL),
       ('Baer',    'Berlinale',    'Germany',  'Berliner Baer(-Gold,Silver,Bronze)',   1951,NULL),
       ('H****','Halliwell`s Film Guide',  'Great Britain',    'four stars',   1983,NULL),
       ('H***','Halliwell`s Film Guide',   'Great Britain',    'three stars',  1983,NULL);
```

3. **Casts table:**

```
INSERT INTO casts (film_id, title, actor, roletype, role)
VALUES ('AA13','Pygmalion', 'Leslie Howard',    'Sci','smug professor \"Higgins\"'),
       ('AA13','Pygmalion',   'Wendy Hiller', 'Inn',  'flower girl \"Eliza\"'),
       ('AA13','Pygmalion',   'Wilfrid Lawson'    ,'Und', 'friend \"Dolittle\"'),
       ('AA13',   'Pygmalion',    'Marie Lohr',   'Und',  'wife \"Mrs.Higgins\"');
```

4. **Categories table:**

```
INSERT INTO categories (code,category)
Values ('Susp', 'thriller'),
       ('CnR', 'cops and robbers'),
       ('Dram',    'drama'),
       ('West',    'western'),
       ('Myst',    'mystery'),
       ('Comd', 'Comedy'),
       ('Ctxx',    'uncategorized');
```

5. **Colorcodes table:**

```
INSERT INTO colorcodes (code, fullname, description)
Values  ('prc','unknown',NULL),
       ('col', 'color','color film, common after 1955'),
       ('bnw', 'black-and-white','b-w film common before 1945'),
       ('sbw','silent','silent black-and-white film'),
       ('cld', 'colored',  'black-and-white film recolored'),
       ('Cart',    'cartoon',  'Cartoons are normally colored');
```

6. **Geography table:**

```
INSERT INTO geography (code, country, adjective)
Values ('Am',   'USA',  'American'),
       ('Br',  'not used', 'British'),
       ('GB',  'Great Britain',    'not used'),
       ('Fr',  'France',   'French'),
       ('Ge',  'Germany',  'German'),
       ('It',  'Italy',    'Italian'),
       ('Ja',  'Japan',    'Japanese'),
       ('Hu',  'Hungary',  'Hungarian'),
       ('Au',  'Australia',    'Australian');
```

7. **Movies table:**

```
INSERT INTO movies (film_id, title, release_year, director, producers, studios, process, category, awards)
VALUES ('AA13', 'Pygmalion' ,1938,  'Asquith'   ,'G.Pascal',NULL    ,   'bnw',  'Comd', 'H****'),
       ('AA14','French Without Tears',    1939,   'Asquith','David Kelly' ,'Paramount'    ,'bnw', 'Ctxx', NULL),
       ('AA16','Quiet Wedding',   1940,   'Asquith',  NULL    ,'Paramount','bnw',NULL,'H***'),
       ('AA20','The Demi-Paradise',   1943,   'Asquith',NULL,NULL,'prc',  'Ctxx',NULL),
       ('LuB20','Pygmalion',1937, 'Aaron','David Kelly','Kinetoscope','bnw','Dram','AA'),
      ('HH15','Twentieth Century',    1934,'Aaron','David Kelly','Paramount','bnw',   'Comd'  ,'H****');
```

8. **People table:**

```
INSERT INTO people (ref_name, codes, d_id, years, last_name, first_name, yob, yodeath,country)
VALUES  ('Aaron',   'D',    'PAa', '1979',  'Aaron',    'Paul', NULL,NULL,'Am'),
       ('Abel',    'D',    'JeA','1971',   'Abel', 'Jeanne',NULL,NULL,'Am'),
       ('Abbott','D','GgA','1929-1958',    'Abbott',   'George','1887',NULL,'Am'),
       ('Abrahams',    'D','xAb',  '1948', 'Abrahams',NULL,NULL,NULL,'Am'),
       ('J.Abrahams',  'D',    'JiA'   ,'1980-1988',   'Abrahams', 'Jim',NULL,NULL,'Am'),
       ('Asquith', 'D',    'AA',   '1928', 'Asquith',  'Anthony `Puffin',  1902,   1968    ,'Br'),
       ('G.Pascal',    'P',    'GPa',  '1940-1945',    'Pascal'    ,'Gabriel', 1894,   1954,   'Hu'),
       ('David Kelly'  ,'P',   NULL,   1993,   'Kelly' ,'David E.',    1956,NULL,  'Am');
```

9. **Roletype table:**

```
INSERT INTO roletypes (roletype, description)
VALUES  ('Adv','adversary'),
       ('Agn','agent'),
       ('Ani','animal'),
       ('Bit','bit role'),
       ('Cam','cameo role'),
       ('Cro', 'crook'),
       ('Grp','group or band'),
       ('Sci'  ,'scientist'),
       ('Inn', 'innocent'),
       ('Und'  ,'undetermined');
```

10. **Studios table:**

```
INSERT INTO studios (studioname, company, city, country, foundeddate, enddate)
VALUES ('Projection-Praxinoscope',NULL,NULL,'France','1881','1882'),
       ('Film Camera',NULL,NULL,'Germany','1889',NULL),
       ('35mm Film Camera',    'Edison',   'New Jersey','USA', '1893',NULL),
       ('Kinetoscope', 'Edison','New Jersey','USA',    '1894',NULL),
       ('Practical Projectors',    'Lumiere Fr.','Lyon',   'France',   '1895',NULL),
       ('Paramount',   'Paramount Corp.','Los Angeles',    'USA',  '1916', '1993');
```

Alter Scripts:

```
ALTER table Movies
Add notes varchar(200);

ALTER table actors
Add Biography varchar(200);
```

This SQL statement adds new columns notes and Biography in tables Movies and actors respectively.

Drop Script:

```
DROP table remakes;
```

This SQL statement removes the table from the database. If there were any keys that are foreign in other tables, the table is not dropped.

Select Script:

1. Find the total number of movies produced by each studio and order the result by the number of movies in descending order.

   Script:

   SELECT s.studioname, COUNT(mn.film_id) AS total_movies

   FROM studios s

   LEFT JOIN movies_normalized mn ON s.studioname = mn.studios
   GROUP BY s.studioname
   ORDER BY total_movies DESC;

Query | Query History

```
1  SELECT s.studioname, COUNT(mn.film_id) AS total_movies
2  FROM studios s
3  LEFT JOIN movies_normalized mn ON s.studioname = mn.studios
4  GROUP BY s.studioname
5  ORDER BY total_movies DESC;
```

Data Output | Messages | Notifications

	studioname [PK] character varying (50)	total_movies bigint
1	Warners	454
2	MGM	443
3	Fox	411
4	Paramount	400
5	Shamley	367
6	Universal	298
7	Columbia	281
8	RKO	194
9	U.A.	188
10	Orion	41
11	Miramax	36
12	AIP	35
13	Republic	31
14	Monogram	30
15	Goldwyn	24
16	GFD	23
17	Carosse	19

Total rows: 175 of 175 | Query complete 00:00:00.222

2. Retrieve the most common role type for each actor, along with the count of movies they played that role:

```
SELECT
  actors.stagename,
  roletypes.roletype AS most_common_role,
  COUNT(*) AS movie_count
FROM actors
INNER JOIN casts ON actors.stagename = casts.actor
INNER JOIN roletypes ON casts.roletype = roletypes.roletype
GROUP BY actors.stagename, roletypes.roletype
HAVING COUNT(*) = (
  SELECT MAX(movie_count)
```

```
  FROM (
    SELECT COUNT(*) AS movie_count
    FROM casts
    WHERE casts.actor = actors.stagename
    GROUP BY casts.roletype
  ) AS role_counts
)
ORDER BY actors.stagename;
```

Query | Query History

```sql
SELECT
  actors.stagename,
  roletypes.roletype AS most_common_role,
  COUNT(*) AS movie_count
FROM actors
INNER JOIN casts ON actors.stagename = casts.actor
INNER JOIN roletypes ON casts.roletype = roletypes.roletype
GROUP BY actors.stagename, roletypes.roletype
HAVING COUNT(*) = (
  SELECT MAX(movie_count)
  FROM (
    SELECT COUNT(*) AS movie_count
    FROM casts
    WHERE casts.actor = actors.stagename
    GROUP BY casts.roletype
  ) AS role_counts
)
ORDER BY actors.stagename;
```

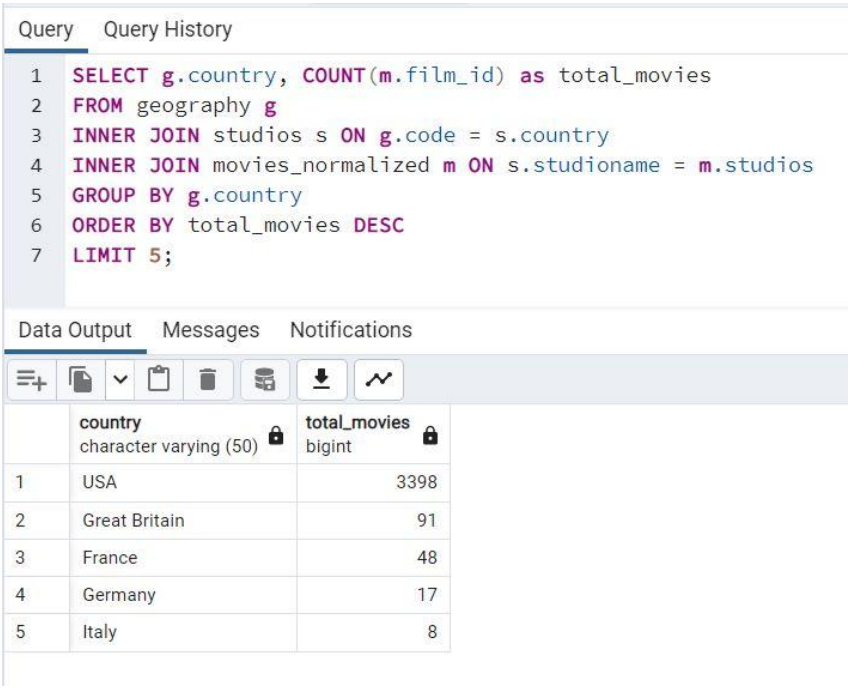
Data Output | Messages | Notifications

	stagename character varying (50)	most_common_role character varying (10)	movie_count bigint
1	A.E. Matthews	Und	3
2	Adam Faith	Und	2
3	Adam West	Und	3
4	Addison Richards	Und	2
5	Adele Mara	Und	1
6	Adeline deWalt Reynolds	Und	1
7	Adolfo Celi	Und	4
8	Adolphe Menjou	Und	20
9	Adrienne Barbeau	Und	5

Total rows: 1000 of 3885 | Query complete 00:00:20.652

3. Get the top 5 countries with the most number of movies produced, along with the number of movies produced and the name of the country:

```
SELECT g.country, COUNT(m.film_id) as total_movies
FROM geography g
INNER JOIN studios s ON g.code = s.country
INNER JOIN movies_normalized m ON s.studioname = m.studios
GROUP BY g.country
ORDER BY total_movies DESC
LIMIT 5;
```



Query | Query History

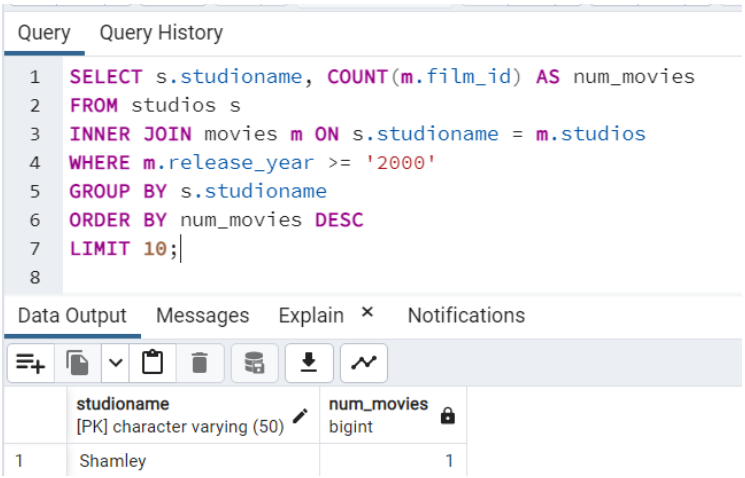
```
1  SELECT g.country, COUNT(m.film_id) as total_movies
2  FROM geography g
3  INNER JOIN studios s ON g.code = s.country
4  INNER JOIN movies_normalized m ON s.studioname = m.studios
5  GROUP BY g.country
6  ORDER BY total_movies DESC
7  LIMIT 5;
```

Data Output | Messages | Notifications

	country character varying (50)	total_movies bigint
1	USA	3398
2	Great Britain	91
3	France	48
4	Germany	17
5	Italy	8

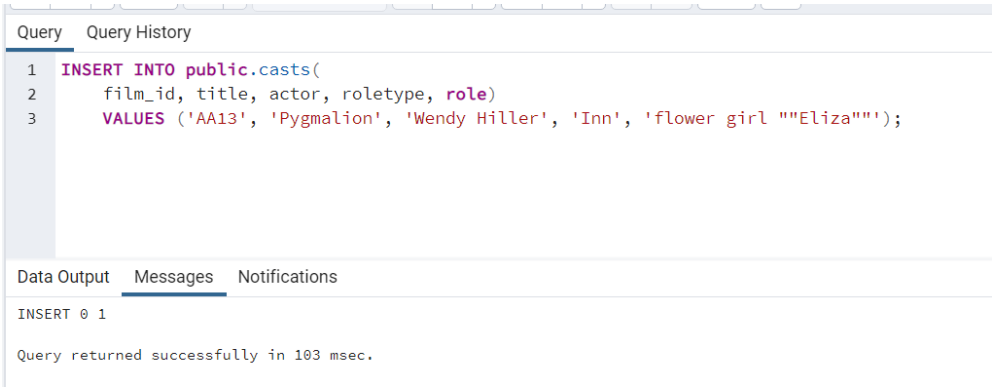
4. Query to select the top 10 studios with the highest number of movies released in the 21st century:

```
SELECT s.studioname, COUNT(m.film_id) AS num_movies
FROM studios s
INNER JOIN movies m ON s.studioname = m.studios
WHERE m.release_year >= '2000'
GROUP BY s.studioname
ORDER BY num_movies DESC
LIMIT 10;
```



Insert Script:

INSERT INTO public.casts(
film_id, title, actor, roletype, role)
VALUES ('AA13', 'Pygmalion', 'Wendy Hiller', 'Inn', 'flower girl ""Eliza""');



Update Script:

UPDATE public.casts

SET title='One Movie', roletype='Inn'

WHERE film_id = 'AA13'

This SQL statement updates the row with USA and wherever required.

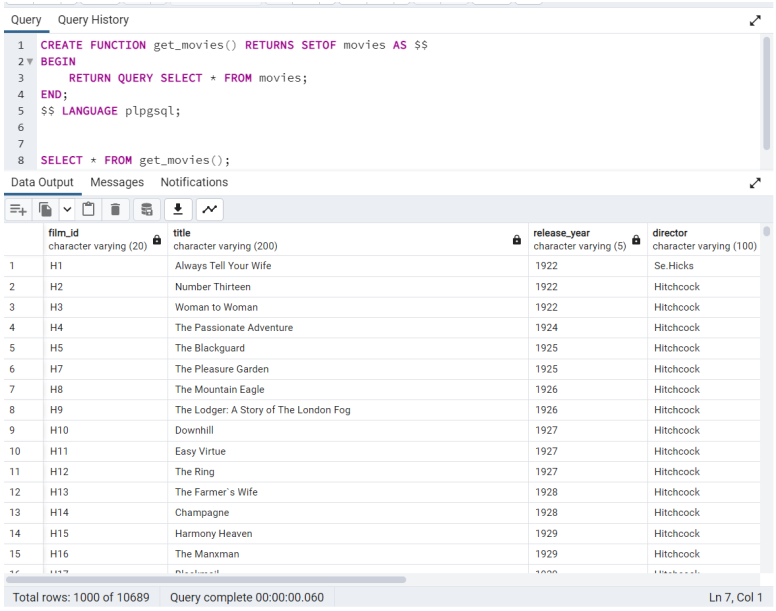
Delete Scripts:

This SQL statement deletes rows with ‘Und’ from categories and anywhere in the database as we are using Cascade on delete.

We are using cascade on delete and on update as the data is dependent on other tables data so we need to maintain data consistency across the database.

Functions:

```
CREATE FUNCTION get_movies() RETURNS SETOF movies AS $$
BEGIN
   RETURN QUERY SELECT * FROM movies;
END;
$$ LANGUAGE plpgsql;

SELECT * FROM get_movies();
```



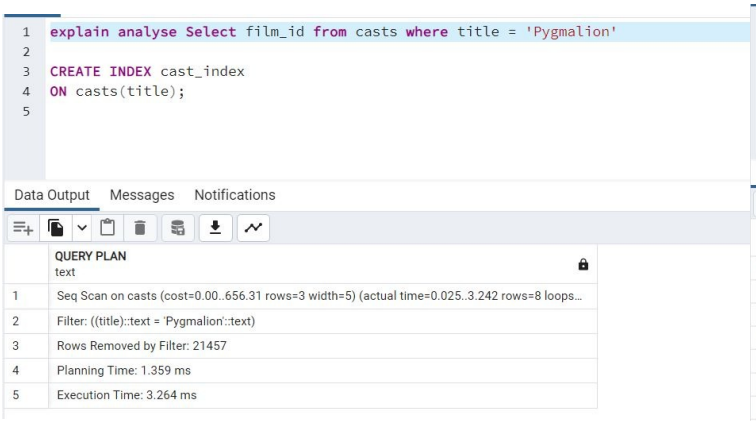
```
CREATE FUNCTION get_movies() RETURNS SETOF movies AS $$
BEGIN
    RETURN QUERY SELECT * FROM movies;
END;
$$ LANGUAGE plpgsql;


SELECT * FROM get_movies();
```

	film_id character varying (20)	title character varying (200)	release_year character varying (5)	director character varying (100)
1	H1	Always Tell Your Wife	1922	Se.Hicks
2	H2	Number Thirteen	1922	Hitchcock
3	H3	Woman to Woman	1922	Hitchcock
4	H4	The Passionate Adventure	1924	Hitchcock
5	H5	The Blackguard	1925	Hitchcock
6	H7	The Pleasure Garden	1925	Hitchcock
7	H8	The Mountain Eagle	1926	Hitchcock
8	H9	The Lodger: A Story of The London Fog	1926	Hitchcock
9	H10	Downhill	1927	Hitchcock
10	H11	Easy Virtue	1927	Hitchcock
11	H12	The Ring	1927	Hitchcock
12	H13	The Farmer`s Wife	1928	Hitchcock
13	H14	Champagne	1928	Hitchcock
14	H15	Harmony Heaven	1929	Hitchcock
15	H16	The Manxman	1929	Hitchcock

Total rows: 1000 of 10689 | Query complete 00:00:00.060 | Ln 7, Col 1

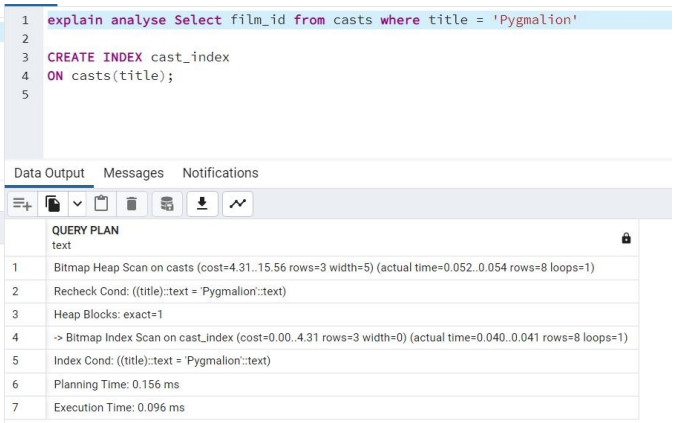
Indexing:

Indexing is a way to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed. It is a data structure technique which is used to quickly locate and access the data in a database.



```
explain analyse Select film_id from casts where title = 'Pygmalion'

CREATE INDEX cast_index
ON casts(title);
```

QUERY PLAN text
Seq Scan on casts (cost=0.00..656.31 rows=3 width=5) (actual time=0.025..3.242 rows=8 loops...
Filter: ((title)::text = 'Pygmalion'::text)
Rows Removed by Filter: 21457
Planning Time: 1.359 ms
Execution Time: 3.264 ms

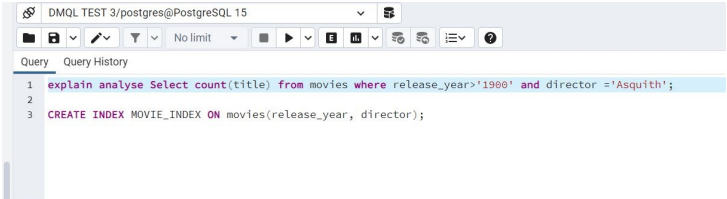


```
explain analyse Select film_id from casts where title = 'Pygmalion'

CREATE INDEX cast_index
ON casts(title);
```

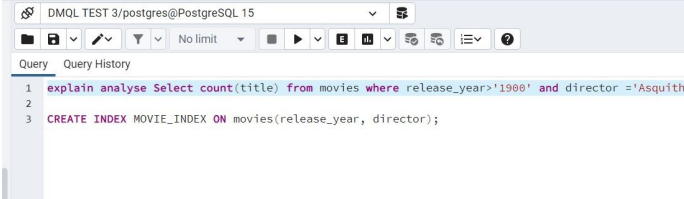
QUERY PLAN text
Bitmap Heap Scan on casts (cost=4.31..15.56 rows=3 width=5) (actual time=0.052..0.054 rows=8 loops=1)
Recheck Cond: ((title)::text = 'Pygmalion'::text)
Heap Blocks: exact=1
-> Bitmap Index Scan on cast_index (cost=0.00..4.31 rows=3 width=0) (actual time=0.040..0.041 rows=8 loops=1)
Index Cond: ((title)::text = 'Pygmalion'::text)
Planning Time: 0.156 ms
Execution Time: 0.096 ms

An Index is created as cast_index for the increase of performance so that retrieval is faster. It is created for the table of casts with the column of title. We can clearly see that planning time and execution time before creating index is 1.359 ms and 3.264 ms respectively. After creating the index it has decreased and the values are 0.156 ms and 0.096 ms respectively.There is significant change when the retrieval is done with the help of indexing.

Similarly an index is created on the movies table with the columns of release year and director. There is a change in Execution time. The execution time before and after creating index is shown below:



```
explain analyse Select count(title) from movies where release_year>'1900' and director ='Asquith';

CREATE INDEX MOVIE_INDEX ON movies(release_year, director);
```



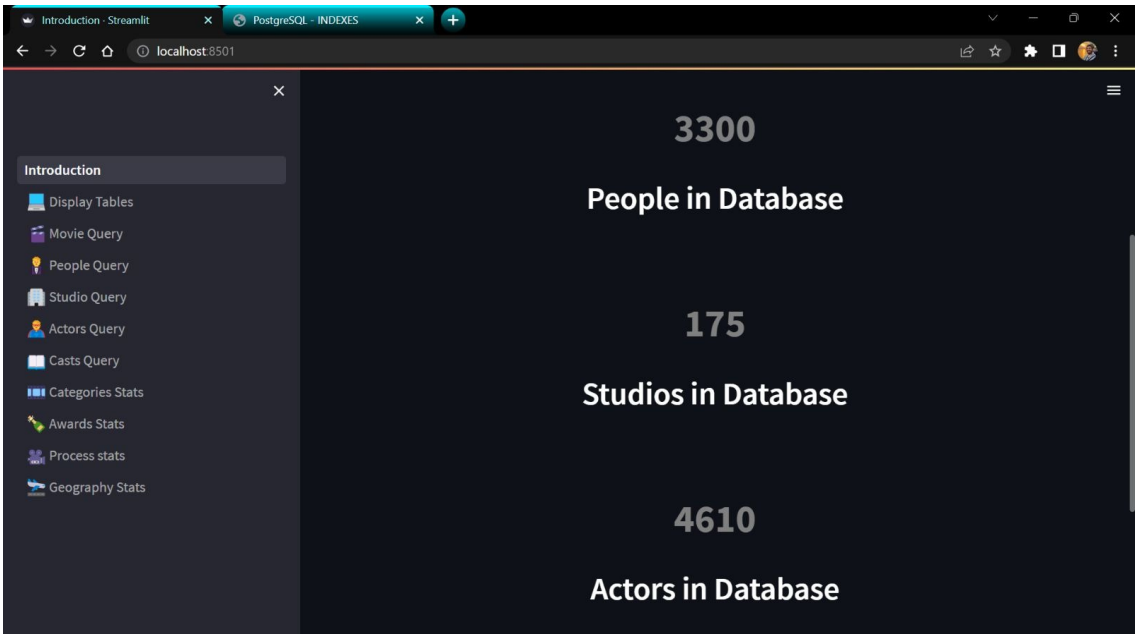
```
explain analyse Select count(title) from movies where release_year>'1900' and director ='Asquith

CREATE INDEX MOVIE_INDEX ON movies(release_year, director);
```

User Interface: A web application has been developed using Streamlit, an open-source Python app framework, to provide an interactive real-time user experience. To connect to PostgreSQL, the application utilizes the Psycopg2 library in Python.

The application features a menu panel containing options for creating, reading, updating, deleting, and querying editor operations. Users can perform dynamic operations by selecting options from the dropdown list and entering desired values in the text box. The query editor allows for custom queries, and the results are displayed in the results tab with visualized graphs.

Working Instructions for UI:

1. Open command prompt and go to the project folder with cd command.
2. Run the app.py file using the command “streamlit run app.py”
3. A local host URL is created, and the web app opens in the browser as shown in below figure.



4. In UI, we have created following as below
    i. Display Tables
    ii. Movie Query
    iii. People Query
    iv. Studio Query
    v. Actors Query

vi. Casts Query

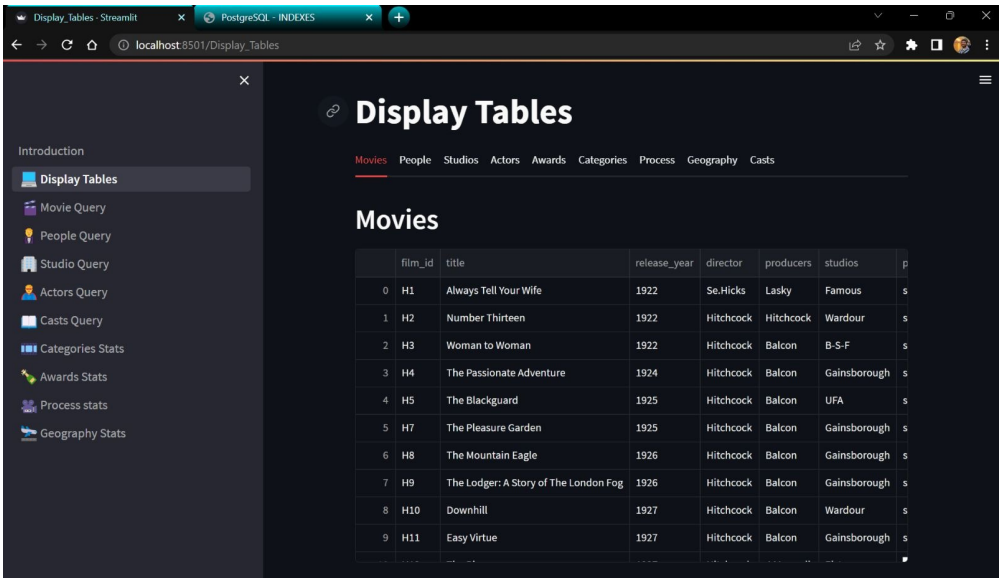
vii. Category Stats

viii. Awards Stats

ix. Process Stats

x. Geography Stats

a. **Display Table**: It displays the sample data from all the tables in the Database upto 50 data points so that the end user will be getting the clear cut idea of how the data is present so that end user can query.



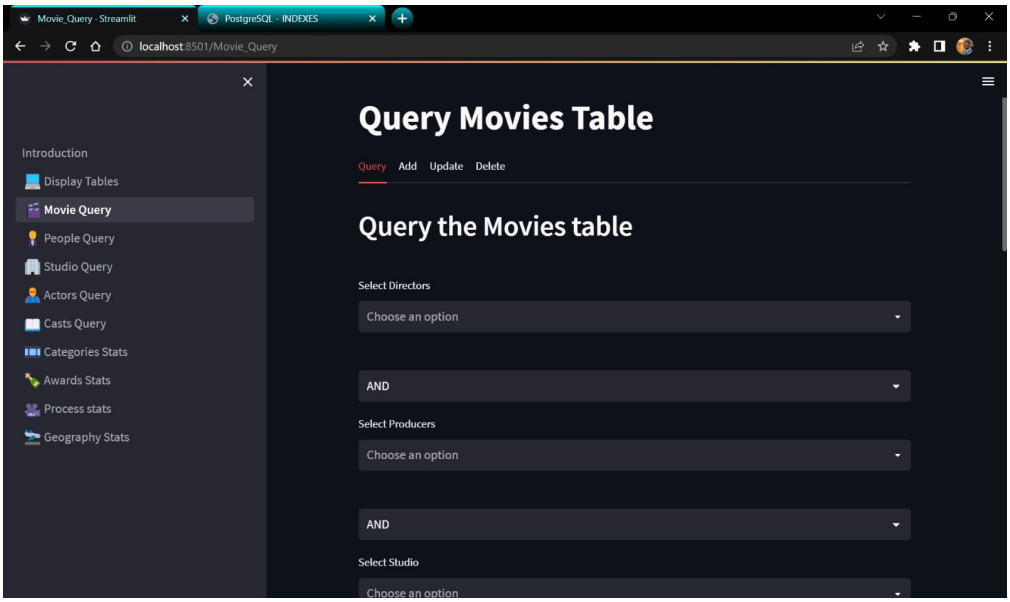
b. **Query Movies Table:** This page helps us to query about Movie Table, we have 4 tabs which are on the movie table that are Query, Add, Update and Delete. For this we have used CRUD Operations at the backend of the Database.

i. Query: Select Query based on the inputs of the data provided by the end user, website gives the results based on the input.

ii. Add: Based on the inputs of the data provided by the end user, based on the data, data gets inserted at the backend database.

iii. Update: On the values of the end user, the values gets updated to the rows in the database.

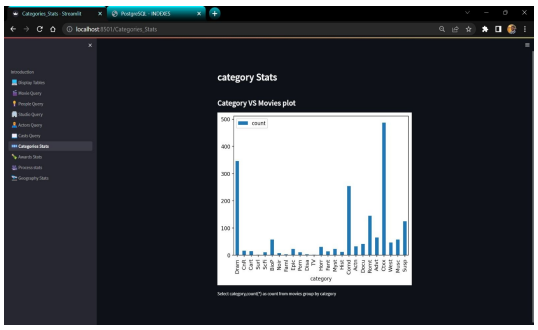
iv. Delete: Based on the inputs of the data, the rows gets deleted given by the end user in the database.

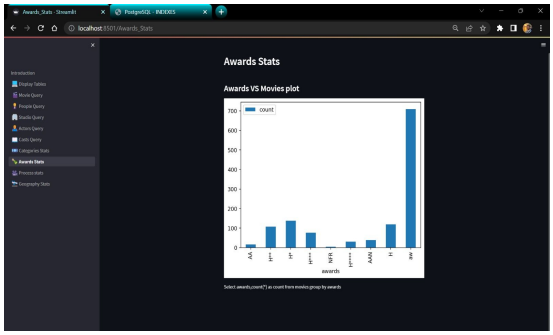


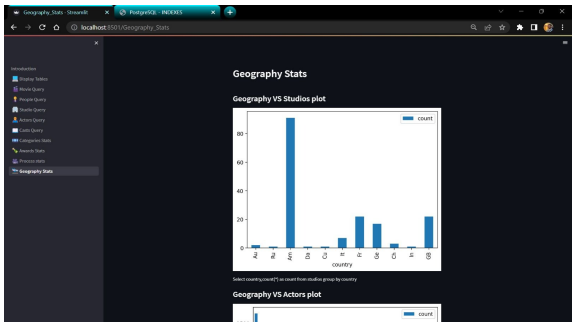
Similarly, we have pages like People, Studios, Actors, Casts table.

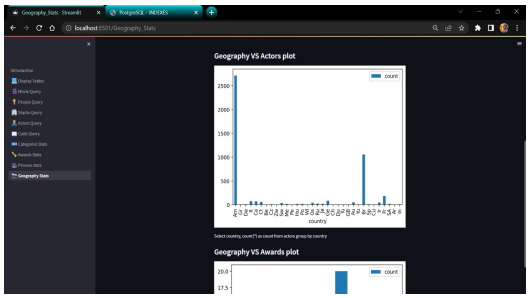
5. Some stats about the data present in the tables:

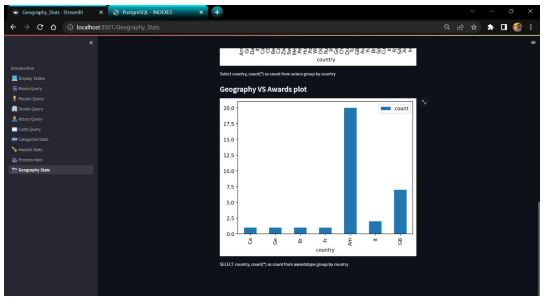
   Category Stats, Awards Stats, Process Stats, and Geography Stats











Dataset:

For this project, we have chosen the “Movie Data Set” from the UCI’s machine learning repository: https://archive.ics.uci.edu/ml/datasets/Movie. It has “real” data allowing to check results for semantic as well as syntactic correctness. This data set contains a list of over 10,000 films including many older, odd, and cult films. There is information on actors, casts, directors, producers, studios, etc. The entries were gradually collected starting about 1975 and are still being updated.

Data Transformation:

The original data is stored in relational form across several files. The current files are in HTML format. In order to store the data in our RDBMS, we have written a Java program to parse these HTML files into corresponding CSV files that will facilitate storing them in PostgreSQL’s tables.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class ConvertHtmlToCsv {

	// Working code to convert UCI's Movie data set from HTML to CSV format
	public static void main(String[] args) {
	String[] files = {"synonyms.html",
	                  "actors.html",
	                  "casts.html",
	                  "remakes.html",
```

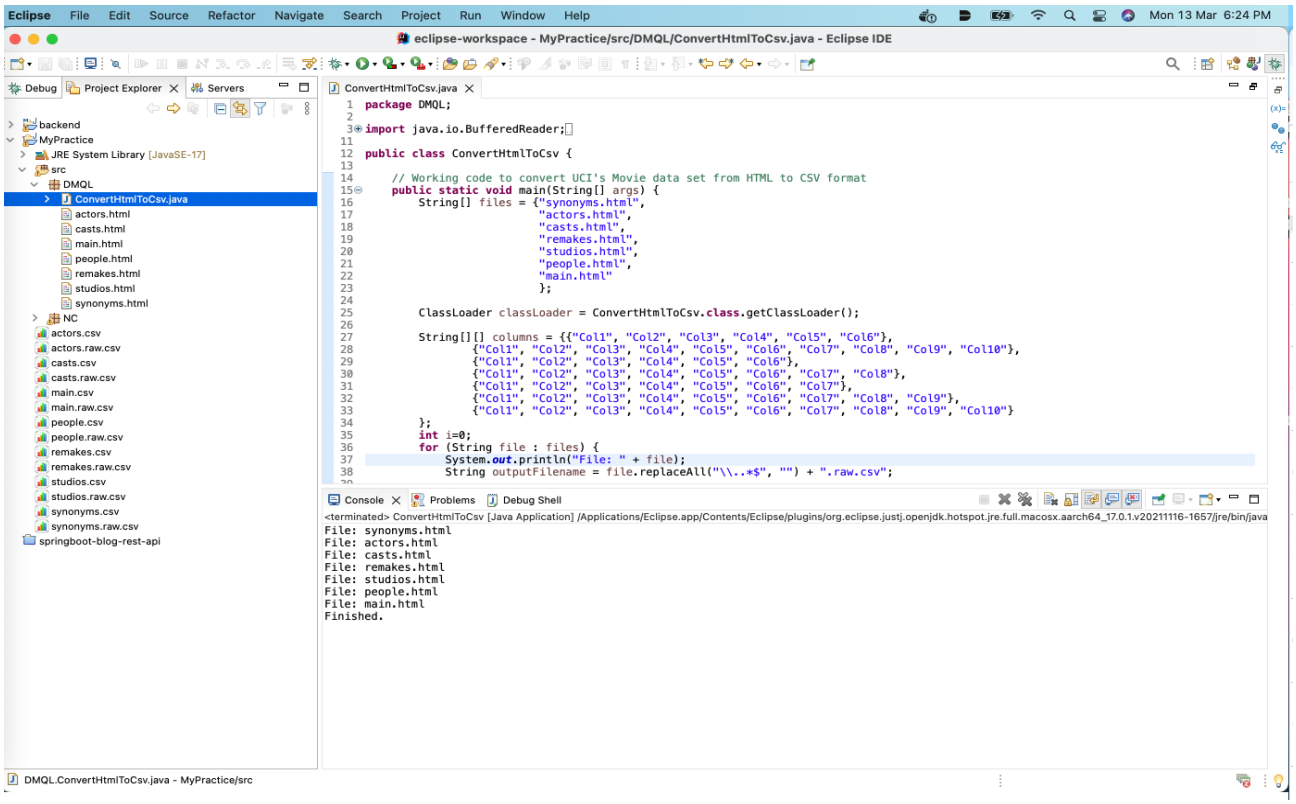
```
                        "studios.html",
                        "people.html",
                        "main.html"
                       };

      ClassLoader classLoader = ConvertHtmlToCsv.class.getClassLoader();

      String[][] columns = {{"Col1", "Col2", "Col3", "Col4", "Col5", "Col6"},
                  {"Col1", "Col2", "Col3", "Col4", "Col5", "Col6", "Col7",
"Col8", "Col9", "Col10"},
                  {"Col1", "Col2", "Col3", "Col4", "Col5", "Col6"},
                  {"Col1", "Col2", "Col3", "Col4", "Col5", "Col6", "Col7",
"Col8"},
                  {"Col1", "Col2", "Col3", "Col4", "Col5", "Col6", "Col7"},
                  {"Col1", "Col2", "Col3", "Col4", "Col5", "Col6", "Col7",
"Col8", "Col9"},
                  {"Col1", "Col2", "Col3", "Col4", "Col5", "Col6", "Col7",
"Col8", "Col9", "Col10"}
      };
      int i=0;
      for (String file : files) {
            System.out.println("File: " + file);
            String outputFilename = file.replaceAll("\\..*$", "") + ".raw.csv";

            InputStream inputStream = classLoader.getResourceAsStream("DMQL/" +
file);

            try (BufferedReader reader = new BufferedReader(new
InputStreamReader(inputStream));
            FileWriter writer = new FileWriter(outputFilename)) {
            String line;
            int tableFlag = 0;
            while ((line = reader.readLine()) != null) {
                   line = removeAccents(line);
                  line = line.replaceAll("\\~", " ");
                  if (line.toLowerCase().contains("<table>")) {
                        tableFlag = 1;
                  }
                  if (tableFlag == 1) {
                        writer.write(line);
                        writer.write("\n");
                  }
                  if (line.toLowerCase().contains("</table>")) {
                        tableFlag = 0;
                  }
            }
            writer.close();
```

```
            writeCsv(outputFilename, columns[i]);
            i++;
            } catch (IOException e) {
            e.printStackTrace();
            }
      }
      System.out.println("Finished.");
      }

      public static String removeAccents(String input) {
       String[] accents = { "\\'a", "\\^a", "\\~a", "\\oa", "\\`a", "\\\"a",
"\\'e", "\\^e", "\\ve", "\\'i", "\\^i",
                  "\\~n", "\\'o", "\\^o", "\\~o", "\\'u", "\\^u", "\\'y",
"\\'c", "\\^c", "\\vc", "\\vd", "\\^h", "\\vl",
                  "\\'r", "\\vr", "\\'s", "\\^s", "\\vs", "\\vt", "\\ou",
"\\^w", "\\^y", "\\'z", "\\vz", "\\`A", "\\`O",
                  "\\`U", "\\`I", "\\`E", "\\\"O", "\\\"U", "\\\"A", "\\\"I",
"\\\"Y", "\\'A", "\\^A", "\\~A", "\\OA",
                  "\\'E", "\\^E", "\\VE", "\\'I", "\\^I", "\\~N", "\\'O",
"\\^O", "\\~O", "\\'U", "\\^U", "\\'Y", "\\'C",
                  "\\^C", "\\VC", "\\VD", "\\^H", "\\VL", "\\'R", "\\VR",
"\\'S", "\\^S", "\\VS", "\\VT", "\\OU", "\\^W",
                  "\\^Y", "\\'Z", "\\VZ" };

       String[] nonAccents = { "á", "â", "ã", "å", "à", "ä", "é", "ê", "ě", "í",
"î", "ñ", "ó", "ô", "õ", "ú", "û",
                  "ý", "ć", "ĉ", "č", "ď", "ĥ", "ĺ", "ŕ", "ř", "ś", "ŝ", "š", "ť",
"ů", "ŵ", "ŷ", "ź", "ž", "Ŕ", "Ň", "Ů",
                  "Ě", "Č", "Ö", "Ü", "Ä", "Ď", "˙", "Á", "Â", "Ă", "Ĺ", "É",
"Ę", "E", "Í", "Î", "Ń", "Ó", "Ô", "Ő", "Ú",
                  "Ű", "Ý", "C", "C", "C", "D", "H", "L", "R", "R", "S", "S",
"Š", "T", "U", "W", "Y", "Z", "Ž" };

       for (int i = 0; i < accents.length; i++) {
             String ac = accents[i].replace("\\", "\\\\");
             Pattern pattern = Pattern.compile(ac);
             Matcher matcher = pattern.matcher(input);
             input = matcher.replaceAll(nonAccents[i]);
       }

       return input;
    }

      private static void writeCsv(String inputFilename, String[] columns) throws
IOException {
      String outputFilename = inputFilename.replaceAll("\\..*$", "") + ".csv";
      try (BufferedReader reader = new BufferedReader(new
```

```
FileReader(inputFilename));
            FileWriter writer = new FileWriter(outputFilename)) {
            String line;
            writer.write(String.join(";", columns) + "\n");
            while ((line = reader.readLine()) != null) {
            String[] values = line.replaceAll("^ *<td>| *</td> *|<tr>|</tr>|\\r",
"").split(";|<td>|</td>");
            if (values.length >= columns.length) {
                  for (int i = 0; i < columns.length; i++) {
                        writer.write("\"" + values[i] + "\"");
                        if (i != columns.length - 1) {
                        writer.write(";");
                        }
                  }
                  writer.write("\n");
            }
            }
      }
      }
}
```

Output:

We will get the converted ".csv" files of the corresponding ".html" files as shown below.

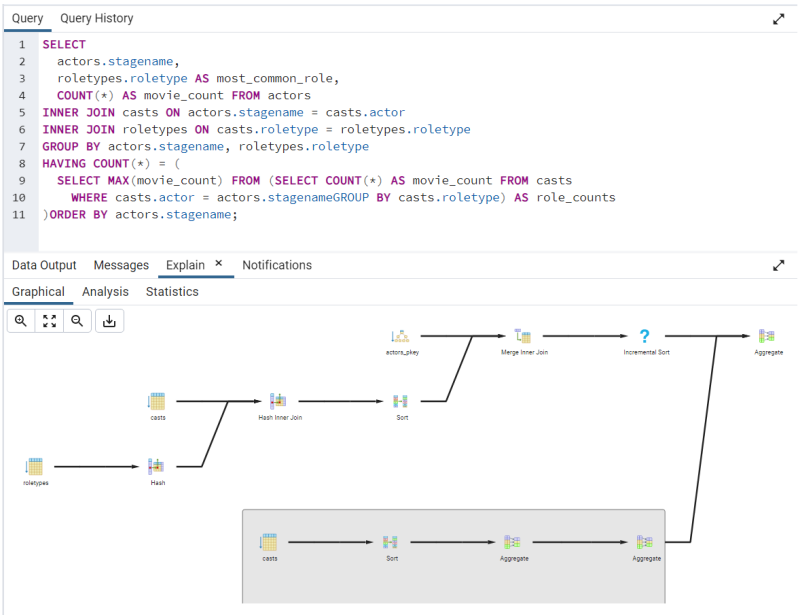


Query Analysis:

In PostgreSQL, query analysis typically involves examining the execution plan generated by the query planner to understand how the query is executed and identifying opportunities for optimization. Here are the steps involved in query analysis in PostgreSQL:

Use the EXPLAIN statement: The EXPLAIN statement is used to generate the query plan for a given SQL statement. This statement can be used to analyze the execution plan for a query, which shows how the query planner will execute the query.
Analyze the output: The output of the EXPLAIN statement provides details about the execution plan, such as the sequence of steps the query planner will use to execute the query, the estimated cost of each step, and the estimated number of rows returned by each step. This information can help identify which parts of the query are taking the longest time to execute, and which indexes or tables are being accessed most frequently.
Optimize the query: Based on the output of the EXPLAIN statement, the query can be optimized by modifying the SQL statement or by creating or modifying indexes or tables to better match the query. For example, adding an index to a frequently accessed column can improve query performance.
Use ANALYZE: The ANALYZE command can be used to update the query planner's statistics about the data in the database. This can help the query planner generate more accurate execution plans and improve query performance.
Overall, query analysis in PostgreSQL involves using EXPLAIN to generate and analyze the execution plan, and then optimizing the query based on the results.

1.



Graphical | Analysis | Statistics

#	Node
1.	→ Aggregate **Filter**: (count(*) = (SubPlan 1))
2.	→ Incremental Sort
3.	→ Merge Inner Join
4.	→ Index Only Scan using actors_pkey on actors as actors
5.	→ Sort
6.	→ Hash Inner Join **Hash Cond**: ((casts.roletype)::text = (roletypes.roletype)::text)
7.	→ Seq Scan on casts as casts
8.	→ Hash
9.	→ Seq Scan on roletypes as roletypes
10.	→ Aggregate
11.	→ Aggregate
12.	→ Sort
13.	→ Seq Scan on casts as casts_1 **Filter**: ((actor)::text = (actors.stagename)::text)

Graphical | Analysis | Statistics

Statistics per Node Type

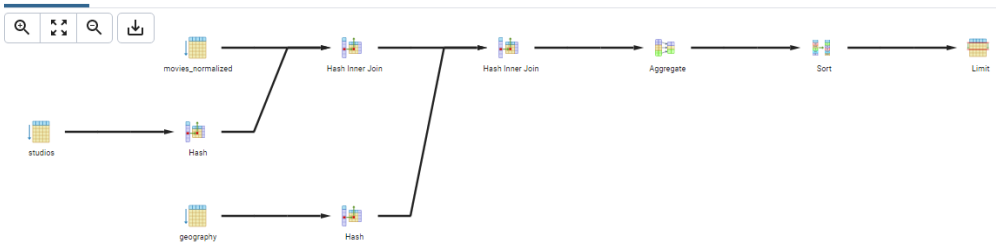
Node type	Count
Aggregate	3
Hash	1
Hash Inner Join	1
Incremental Sort	1
Index Only Scan	1
Merge Inner Join	1
Seq Scan	3
Sort	2

Statistics per Relation

Relation name	Scan count
Node type	**Count**
actors	1
Index Only Scan	1
casts	2
Seq Scan	2
roletypes	1
Seq Scan	1

2. SELECT g.country, COUNT(m.film_id) as total_movies
   FROM geography g
   INNER JOIN studios s ON g.code = s.country
   INNER JOIN movies_normalized m ON s.studioname = m.studios
   GROUP BY g.country
   ORDER BY total_movies DESC
   LIMIT 5;

Graphical Analysis Statistics



Data Output Messages Explain × Notifications

Graphical Analysis Statistics

#	Node	Rows Actual	Loops
1.	→ Limit (rows=5 loops=1)	5	1
2.	→ Sort (rows=5 loops=1)	5	1
3.	→ Aggregate (rows=8 loops=1) **Buckets**: **Batches**: **Memory Usage**: 40 kB	8	1
4.	→ Hash Inner Join (rows=3566 loops=1) **Hash Cond**: ((s.country)::text = (g.code)::text)	3566	1
5.	→ Hash Inner Join (rows=3570 loops=1) **Hash Cond**: ((m.studios)::text = (s.studioname)::text)	3570	1
6.	→ Seq Scan on movies_normalized as m (rows=10689 loops=1)	10689	1
7.	→ Hash (rows=175 loops=1) **Buckets**: 1024 **Batches**: 1 **Memory Usage**: 16 kB	175	1
8.	→ Seq Scan on studios as s (rows=175 loops=1)	175	1
9.	→ Hash (rows=34 loops=1) **Buckets**: 1024 **Batches**: 1 **Memory Usage**: 10 kB	34	1
10.	→ Seq Scan on geography as g (rows=34 loops=1)	34	1

Total rows: 1 of 1 Query complete 00:00:00.071 Ln 7, Col 9

Graphical Analysis Statistics

Statistics per Node Type

Node type	Count
Aggregate	1
Hash	2
Hash Inner Join	2
Limit	1
Seq Scan	3
Sort	1

Statistics per Relation

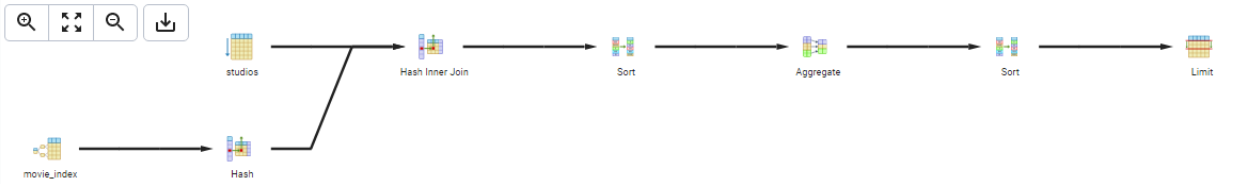
Relation name	Scan count
Node type	**Count**
geography	1
Seq Scan	1
movies_normalized	1
Seq Scan	1
studios	1
Seq Scan	1

Query | Query History

```
1  SELECT s.studioname, COUNT(m.film_id) AS num_movies
2  FROM studios s
3  INNER JOIN movies m ON s.studioname = m.studios
4  WHERE m.release_year >= '2000'
5  GROUP BY s.studioname
6  ORDER BY num_movies DESC
7  LIMIT 10;
8
```

Data Output | Messages | Explain | Notifications

Graphical | Analysis | Statistics

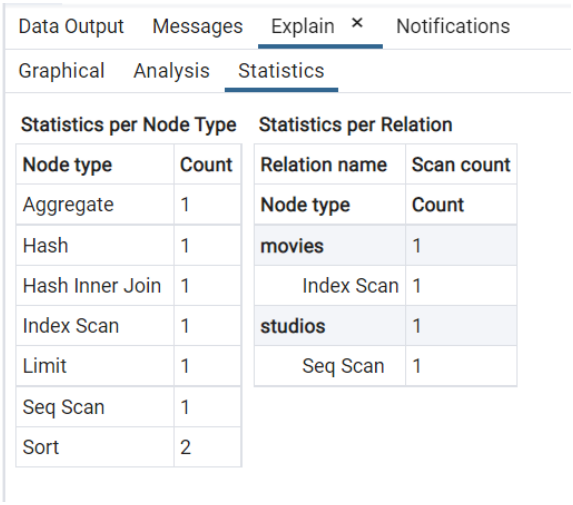


3.

```
SELECT s.studioname, COUNT(m.film_id) AS num_movies
FROM studios s
INNER JOIN movies m ON s.studioname = m.studios
WHERE m.release_year >= '2000'
GROUP BY s.studioname
ORDER BY num_movies DESC
LIMIT 10;
```

Graphical | Analysis | Statistics

#	Node
1.	→ Limit
2.	→ Sort
3.	→ Aggregate
4.	→ Sort
5.	→ Hash Inner Join **Hash Cond**: ((s.studioname)::text = (m.studios)::text)
6.	→ Seq Scan on studios as s
7.	→ Hash
8.	→ Index Scan using movie_index on movies as m **Index Cond**: ((release_year)::text >= '2000'::text)



References

https://archive.ics.uci.edu/ml/datasets/Movie

https://www.postgresql.org/docs/current/tutorial.html

https://www.coollector.com/

https://www.w3schools.com/sql/