CSE 4/560 Milestone 2

MovieGoods

Harsha Vardhan Sivadanam
hsivadan
University at Buffalo
Buffalo, NY, USA
hsivadan@buffalo.edu

Nikhil Kumar Vadigala
nvadigal
University at Buffalo
Buffalo, NY, USA
nvadigal@buffalo.edu

Deepana Nagireddy Gari
deepanan
University at Buffalo
Buffalo, NY, USA
deepanan@buffalo.edu

Abstract- Movie enthusiasts often struggle to find reliable and up-to-date information about movies, such as cast and crew details, release dates, and other details.

The aim of this project is to develop a comprehensive and accurate movie database that stores information about movies and their associated details. The database will include data on movie titles, cast and crew members, release dates, genres, ratings, and other relevant information. The project will involve collecting data from multiple sources and using data cleaning and data integration techniques to ensure that the database is accurate and consistent. The database will also need to be scalable to accommodate new movie releases and updates to existing movie information. The project will also involve developing a user-friendly interface for accessing the database and querying for movie information. The objective of this project is to provide movie enthusiasts with a reliable and comprehensive database that can be used to access information quickly and easily about their favorite movies.

I. INTRODUCTION

Movie enthusiasts are always on the lookout for reliable and up-to-date information about their favorite movies, such as cast and crew details, release dates, ratings, and other relevant information. However, with so many movies being released every year, it can be challenging to keep track of all the necessary details. This is where a comprehensive and accurate movie database comes into play. The aim of this project is to develop a movie database that stores information about movies and their associated details. The database will be designed to be scalable, accommodating new movie releases and updates to existing movie information. With a user-friendly interface, this database will be a valuable resource for movie enthusiasts, providing quick and easy access to information about their favorite movies.

II. TARGET USERS

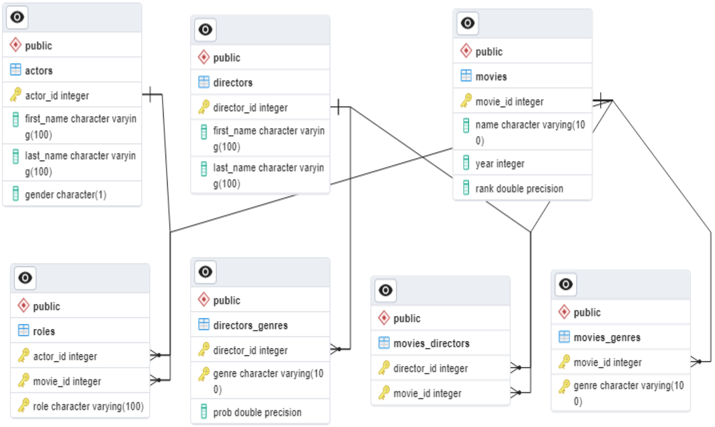
1. User of the database: The target users of the MovieGoods database could be movie enthusiasts, filmmakers, movie critics, researchers, and more importantly people who want to explore different movies. This database is useful for every age group. Mainly the target end users would be the ones between 18-35 age. It can be used by anyone who loves entertainment.

2. Administrator of the database: A team consisting of database administrators, developers, and system analysts are responsible for maintaining, updating, and securing the movie information system database.

3. Real life-life scenario: A real-life scenario for the movie information system could be a popular movie streaming service such as Netflix or Amazon Prime. These services could use the database to recommend movies to their users based on their viewing history and preferences. The database could also be used by movie review websites to provide comprehensive information about a particular movie, including ratings and reviews from various sources, and relevant information about the movie's cast and crew. The database could also be used by cinema halls to promote and screen movies that are popular among their audience.

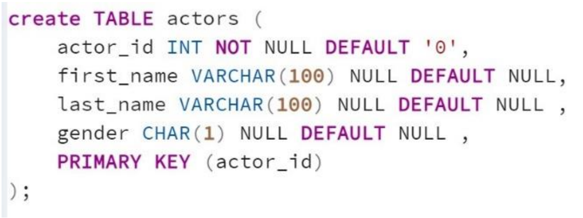
III. ENTITY RELATIONSHIP DIAGRAM

An entity set refers to a group of interconnected entities that process distinct attributes defining their characteristics. ER diagrams employ this concept by identifying properties, and the connections that link them. ER diagrams are primarily utilized to establish the fundamental design of databases.

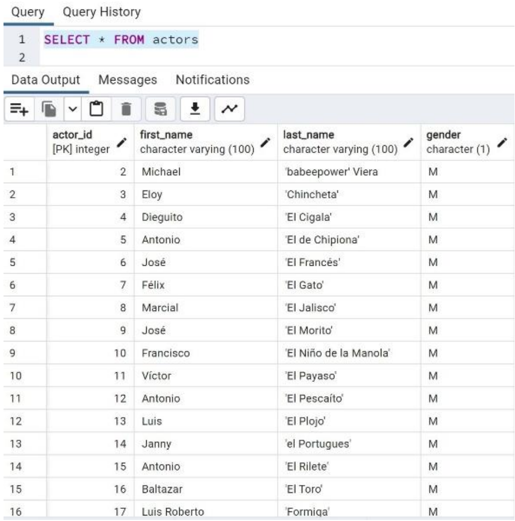


IV. RELATIONS AND SAMPLE DATA

A. Actors Table



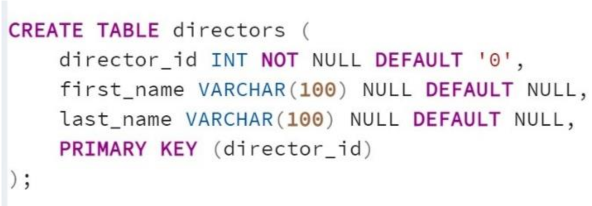
```
create TABLE actors (
    actor_id INT NOT NULL DEFAULT '0',
    first_name VARCHAR(100) NULL DEFAULT NULL,
    last_name VARCHAR(100) NULL DEFAULT NULL ,
    gender CHAR(1) NULL DEFAULT NULL ,
    PRIMARY KEY (actor_id)
);
```



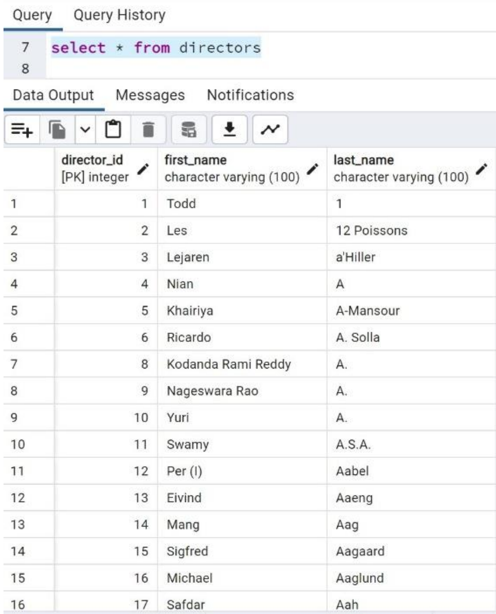
```
SELECT * FROM actors
```

	actor_id [PK] integer	first_name character varying (100)	last_name character varying (100)	gender character (1)
1	2	Michael	'babeepower' Viera	M
2	3	Eloy	'Chincheta'	M
3	4	Dieguito	'El Cigala'	M
4	5	Antonio	'El de Chipiona'	M
5	6	José	'El Francés'	M
6	7	Félix	'El Gato'	M
7	8	Marcial	'El Jalisco'	M
8	9	José	'El Morito'	M
9	10	Francisco	'El Niño de la Manola'	M
10	11	Víctor	'El Payaso'	M
11	12	Antonio	'El Pescaíto'	M
12	13	Luis	'El Plojo'	M
13	14	Janny	'el Portugues'	M
14	15	Antonio	'El Rilete'	M
15	16	Baltazar	'El Toro'	M
16	17	Luis Roberto	'Formiga'	M

B. Directors Table



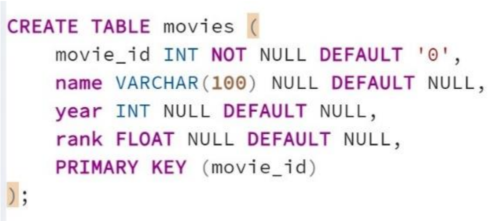
```
CREATE TABLE directors (
    director_id INT NOT NULL DEFAULT '0',
    first_name VARCHAR(100) NULL DEFAULT NULL,
    last_name VARCHAR(100) NULL DEFAULT NULL,
    PRIMARY KEY (director_id)
);
```



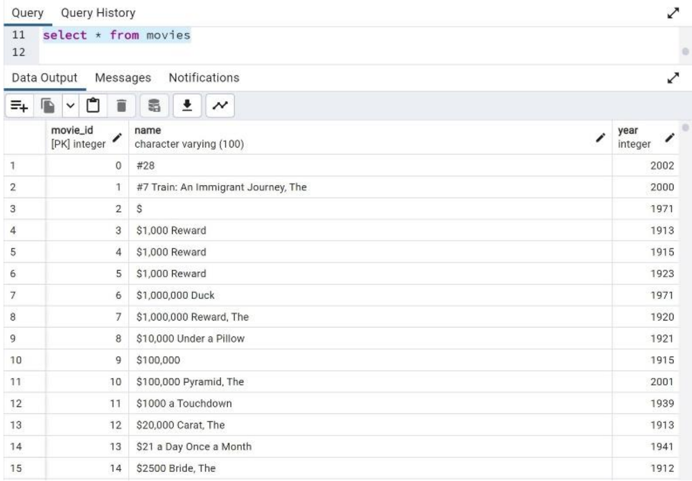
```
select * from directors
```

	director_id [PK] integer	first_name character varying (100)	last_name character varying (100)
1	1	Todd	1
2	2	Les	12 Poissons
3	3	Lejaren	a'Hiller
4	4	Nian	A
5	5	Khairiya	A-Mansour
6	6	Ricardo	A. Solla
7	8	Kodanda Rami Reddy	A.
8	9	Nageswara Rao	A.
9	10	Yuri	A.
10	11	Swamy	A.S.A.
11	12	Per (I)	Aabel
12	13	Eivind	Aaeng
13	14	Mang	Aag
14	15	Sigfred	Aagaard
15	16	Michael	Aaglund
16	17	Safdar	Aah

c. Movies Table



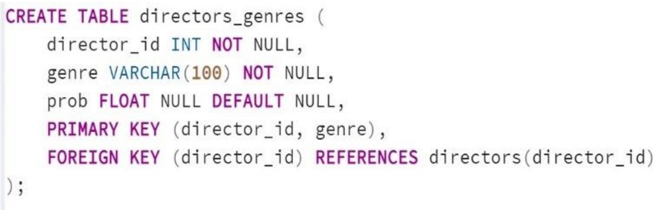
```
CREATE TABLE movies (
    movie_id INT NOT NULL DEFAULT '0',
    name VARCHAR(100) NULL DEFAULT NULL,
    year INT NULL DEFAULT NULL,
    rank FLOAT NULL DEFAULT NULL,
    PRIMARY KEY (movie_id)
);
```



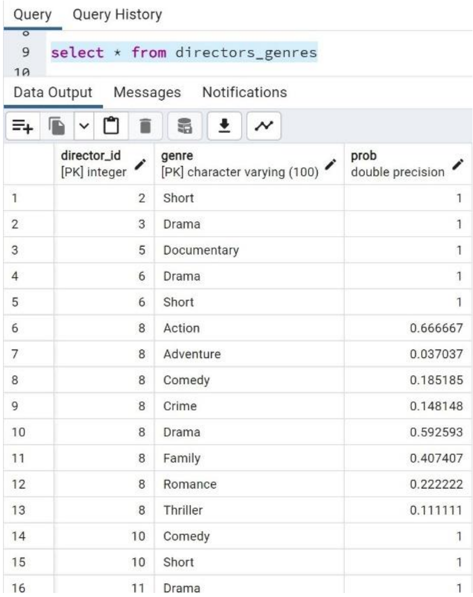
```
select * from movies
```

	movie_id [PK] integer	name character varying (100)	year integer
1	0	#28	2002
2	1	#7 Train: An Immigrant Journey, The	2000
3	2	$	1971
4	3	$1,000 Reward	1913
5	4	$1,000 Reward	1915
6	5	$1,000 Reward	1923
7	6	$1,000,000 Duck	1971
8	7	$1,000,000 Reward, The	1920
9	8	$10,000 Under a Pillow	1921
10	9	$100,000	1915
11	10	$100,000 Pyramid, The	2001
12	11	$1000 a Touchdown	1939
13	12	$20,000 Carat, The	1913
14	13	$21 a Day Once a Month	1941
15	14	$2500 Bride, The	1912

D. Directors Genres Table



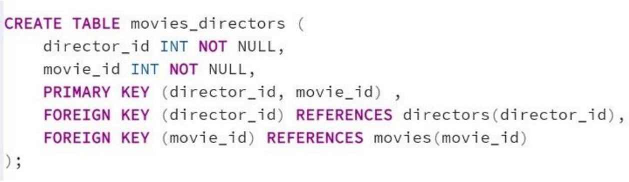
```
CREATE TABLE directors_genres (
    director_id INT NOT NULL,
    genre VARCHAR(100) NOT NULL,
    prob FLOAT NULL DEFAULT NULL,
    PRIMARY KEY (director_id, genre),
    FOREIGN KEY (director_id) REFERENCES directors(director_id)
);
```



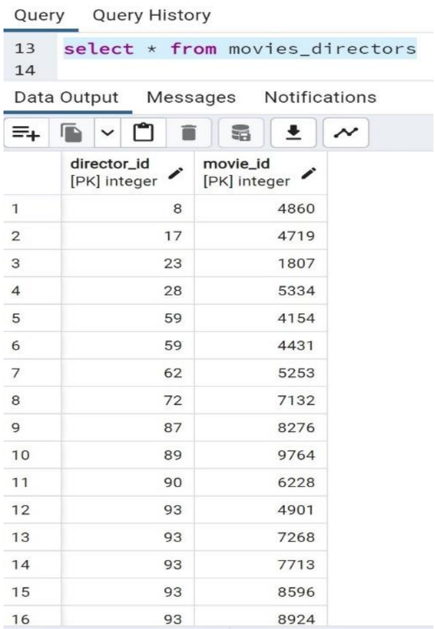
```
select * from directors_genres
```

	director_id [PK] integer	genre [PK] character varying (100)	prob double precision
1	2	Short	1
2	3	Drama	1
3	5	Documentary	1
4	6	Drama	1
5	6	Short	1
6	8	Action	0.666667
7	8	Adventure	0.037037
8	8	Comedy	0.185185
9	8	Crime	0.148148
10	8	Drama	0.592593
11	8	Family	0.407407
12	8	Romance	0.222222
13	8	Thriller	0.111111
14	10	Comedy	1
15	10	Short	1
16	11	Drama	1

E. Movies Directors Table



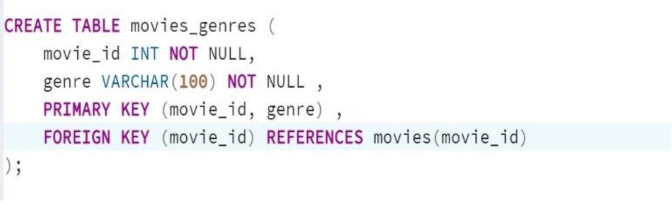
```
CREATE TABLE movies_directors (
    director_id INT NOT NULL,
    movie_id INT NOT NULL,
    PRIMARY KEY (director_id, movie_id) ,
    FOREIGN KEY (director_id) REFERENCES directors(director_id),
    FOREIGN KEY (movie_id) REFERENCES movies(movie_id)
);
```



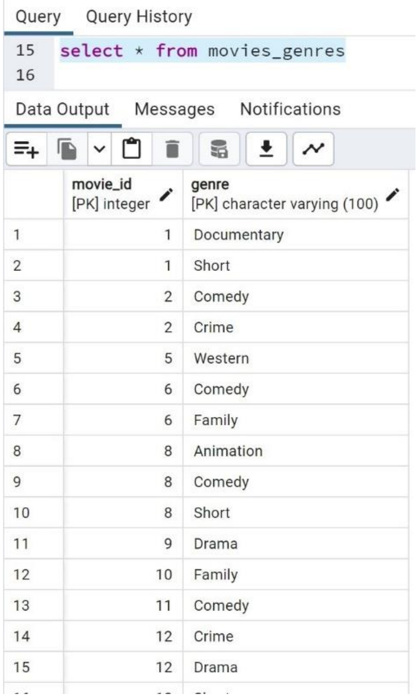
```
select * from movies_directors
```

	director_id [PK] integer	movie_id [PK] integer
1	8	4860
2	17	4719
3	23	1807
4	28	5334
5	59	4154
6	59	4431
7	62	5253
8	72	7132
9	87	8276
10	89	9764
11	90	6228
12	93	4901
13	93	7268
14	93	7713
15	93	8596
16	93	8924

F. Movies Genres Table



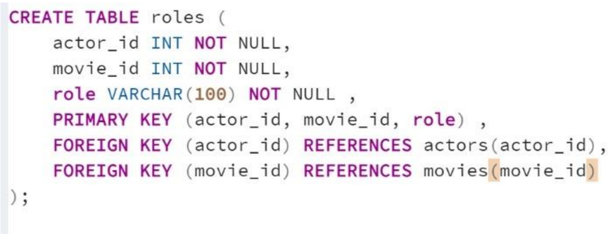
```
CREATE TABLE movies_genres (
    movie_id INT NOT NULL,
    genre VARCHAR(100) NOT NULL ,
    PRIMARY KEY (movie_id, genre) ,
    FOREIGN KEY (movie_id) REFERENCES movies(movie_id)
);
```



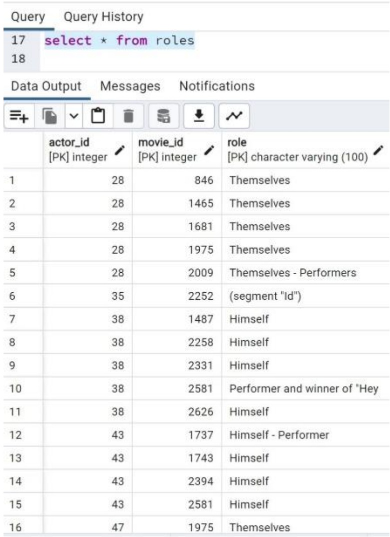
```
select * from movies_genres
```

	movie_id [PK] integer	genre [PK] character varying (100)
1	1	Documentary
2	1	Short
3	2	Comedy
4	2	Crime
5	5	Western
6	6	Comedy
7	6	Family
8	8	Animation
9	8	Comedy
10	8	Short
11	9	Drama
12	10	Family
13	11	Comedy
14	12	Crime
15	12	Drama

G. Roles Table



```
CREATE TABLE roles (
    actor_id INT NOT NULL,
    movie_id INT NOT NULL,
    role VARCHAR(100) NOT NULL ,
    PRIMARY KEY (actor_id, movie_id, role) ,
    FOREIGN KEY (actor_id) REFERENCES actors(actor_id),
    FOREIGN KEY (movie_id) REFERENCES movies(movie_id)
);
```



```
select * from roles
```

	actor_id [PK] integer	movie_id [PK] integer	role [PK] character varying (100)
1	28	846	Themselves
2	28	1465	Themselves
3	28	1681	Themselves
4	28	1975	Themselves
5	28	2009	Themselves - Performers
6	35	2252	(segment "Id")
7	38	1487	Himself
8	38	2258	Himself
9	38	2331	Himself
10	38	2581	Performer and winner of 'Hey
11	38	2626	Himself
12	43	1737	Himself - Performer
13	43	1743	Himself
14	43	2394	Himself
15	43	2581	Himself
16	47	1975	Themselves

V. CONSTRAINTS

Primary Keys:

actors table: actor_id
directors table: director_id
movies table: movie_id
directors_genres: genre
movies_genres: genre
roles: role

Foreign Keys:

directors_genres: director_id
movies_directors: director_id, movie_id
movies_genres: movie_id
roles: actor_id, movie_id

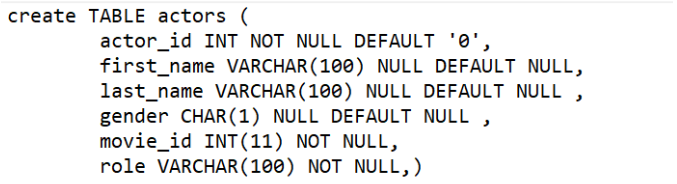
VI. NORMALIZATION

Normalization is the process of organizing data in a database to reduce redundancy and improve data integrity. It involves dividing a large table into smaller, more manageable tables and establishing relationships between them.

The normalization process involves applying a set of rules called Normal Forms to ensure that the database is structured in the most efficient and effective way.

We have 3 tables initially. All the tables are already in 1NF, So, we need to check from there.

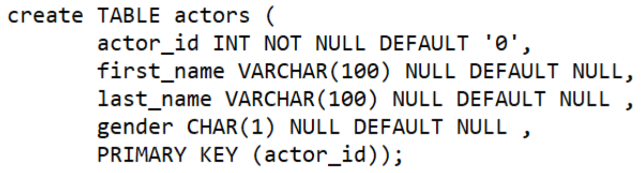
A. Consider the below table of actors:



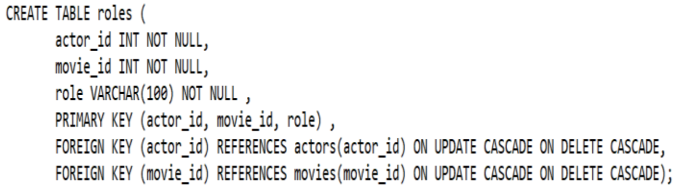
```
create TABLE actors (
        actor_id INT NOT NULL DEFAULT '0',
        first_name VARCHAR(100) NULL DEFAULT NULL,
        last_name VARCHAR(100) NULL DEFAULT NULL ,
        gender CHAR(1) NULL DEFAULT NULL ,
        movie_id INT(11) NOT NULL,
        role VARCHAR(100) NOT NULL,)
```

The table is not in 2NF.

The issue with this table is that the attribute "role" is dependent on both actor_id and movie_id, but only actor_id is part of the primary key. This creates a partial dependency, as a non-key attribute is dependent on only a part of the primary key. So, we split the table to obtain normalization.



```
create TABLE actors (
       actor_id INT NOT NULL DEFAULT '0',
       first_name VARCHAR(100) NULL DEFAULT NULL,
       last_name VARCHAR(100) NULL DEFAULT NULL ,
       gender CHAR(1) NULL DEFAULT NULL ,
       PRIMARY KEY (actor_id));
```



```
CREATE TABLE roles (
      actor_id INT NOT NULL,
      movie_id INT NOT NULL,
      role VARCHAR(100) NOT NULL ,
      PRIMARY KEY (actor_id, movie_id, role) ,
      FOREIGN KEY (actor_id) REFERENCES actors(actor_id) ON UPDATE CASCADE ON DELETE CASCADE,
      FOREIGN KEY (movie_id) REFERENCES movies(movie_id) ON UPDATE CASCADE ON DELETE CASCADE);
```

This separates the role attribute into a separate table with both actor_id and movie_id as the primary key, thereby removing the partial dependency, so is in 2nf.

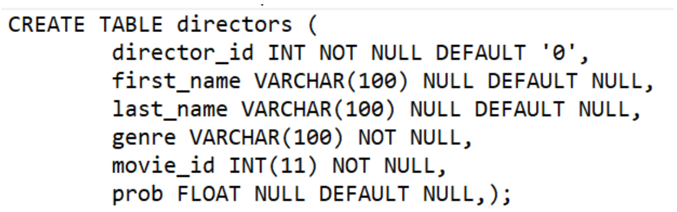
This schema is in 3NF since it satisfies the following conditions:

- Every non-key attribute is dependent on the primary key.
- There are no transitive dependencies between non-key attributes.

In this schema, roles is dependent only on the composite primary key (actor_id, movie_id), and actors has no non-key attributes, so all three conditions are satisfied.

The above schema is in BCNF. All the tables have only one candidate key and each non-key attribute is dependent on the candidate key. There are no non-trivial functional dependencies between non-key attributes.

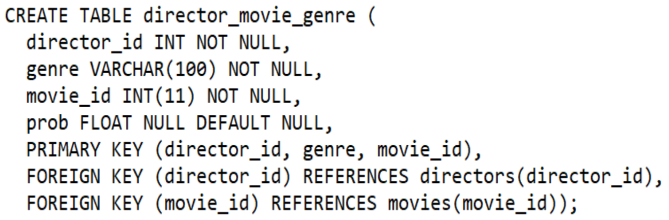
B. Consider the following table of directors:



```
CREATE TABLE directors (
        director_id INT NOT NULL DEFAULT '0',
        first_name VARCHAR(100) NULL DEFAULT NULL,
        last_name VARCHAR(100) NULL DEFAULT NULL,
        genre VARCHAR(100) NOT NULL,
        movie_id INT(11) NOT NULL,
        prob FLOAT NULL DEFAULT NULL,);
```

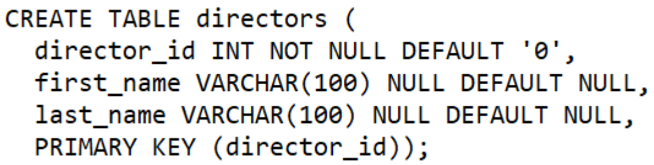
The table is not in 2NF because it has a composite primary key (director_id, genre, movie_id) and a non-key attribute (prob) that depends only on the partial key (director_id, genre).

To bring the table into 2NF, we need to remove the partial dependency by creating a new table for director-genre-movie relationship:



```
CREATE TABLE director_movie_genre (
  director_id INT NOT NULL,
  genre VARCHAR(100) NOT NULL,
  movie_id INT(11) NOT NULL,
  prob FLOAT NULL DEFAULT NULL,
  PRIMARY KEY (director_id, genre, movie_id),
  FOREIGN KEY (director_id) REFERENCES directors(director_id),
  FOREIGN KEY (movie_id) REFERENCES movies(movie_id));
```

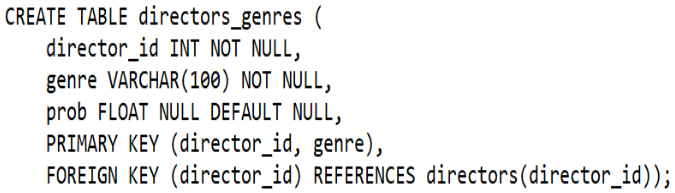
Then, we can remove the genre and movie_id columns from the directors table:



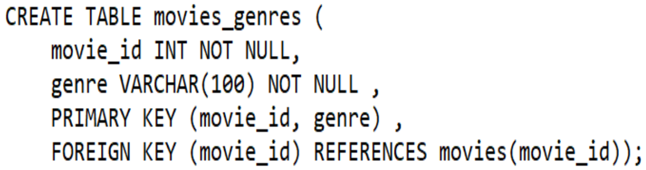
```
CREATE TABLE directors (
  director_id INT NOT NULL DEFAULT '0',
  first_name VARCHAR(100) NULL DEFAULT NULL,
  last_name VARCHAR(100) NULL DEFAULT NULL,
  PRIMARY KEY (director_id));
```

But the director_movie_genre table as defined in the above is not in 2NF because it has a composite primary key (director_id, movie_id, genre) and there is a functional dependency between director_id and genre. In other words, for a given director_id, the genre value is dependent only on the director_id and not on the entire composite key.

To bring the director_movie_genre table to 2NF, we can split it into two tables: directors_genres and movies_genres.



```
CREATE TABLE directors_genres (
    director_id INT NOT NULL,
    genre VARCHAR(100) NOT NULL,
    prob FLOAT NULL DEFAULT NULL,
    PRIMARY KEY (director_id, genre),
    FOREIGN KEY (director_id) REFERENCES directors(director_id));
```



```
CREATE TABLE movies_genres (
    movie_id INT NOT NULL,
    genre VARCHAR(100) NOT NULL ,
    PRIMARY KEY (movie_id, genre) ,
    FOREIGN KEY (movie_id) REFERENCES movies(movie_id));
```

The directors_genres table has a composite primary key (director_id, genre) and no partial dependencies, and the movies_genres table has a composite primary key (movie_id, genre) and no partial dependencies, so both tables are in 2NF.

All the attributes are atomic; hence table is in 1NF.

There is no partial dependency, hence table is in 2NF.

There is no transitive dependency, so table is in 3NF.

Here all the attributes of a relation can be determined for this relation. And all the FDs are non-trivial. Hence the relation is in BCNF.

C. Consider the following table of movies:

```
CREATE TABLE movies (
        movie_id INT NOT NULL DEFAULT '0',
        name VARCHAR(100) NULL DEFAULT NULL,
        year INT NULL DEFAULT NULL,
        rank FLOAT NULL DEFAULT NULL,
        director_id INT(11) NOT NULL,
        genre VARCHAR(100) NOT NULL ,
        actor_id INT(11) NOT NULL,
        role VARCHAR(100) NOT NULL,);
```

The above schema is not in 2NF because it contains partial dependencies. Specifically, the columns director_id, genre, actor_id, and role are dependent on the movie_id, but not on each other. To bring this table to 2NF, we need to split it into multiple tables, each with a single theme.

```
CREATE TABLE movies (
        movie_id INT NOT NULL DEFAULT '0',
        name VARCHAR(100) NULL DEFAULT NULL,
        year INT NULL DEFAULT NULL,
        rank FLOAT NULL DEFAULT NULL,
        PRIMARY KEY (movie_id));
```

```
CREATE TABLE movies_directors (
     director_id INT NOT NULL,
     movie_id INT NOT NULL,
     PRIMARY KEY (director_id, movie_id) ,
     FOREIGN KEY (director_id) REFERENCES directors(director_id) ON UPDATE CASCADE ON DELETE CASC
     FOREIGN KEY (movie_id) REFERENCES movies(movie_id) ON UPDATE CASCADE ON DELETE CASCADE);
```

The new schema appears that the tables are in 3NF and BCNF.

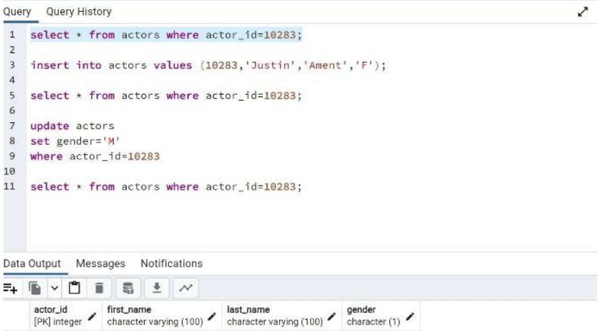
To be in 3NF, a table must meet the following requirements:

- Every non-key attribute is dependent on the primary key.
- All the tables in the given schema have a primary key, and all non-key attributes are dependent on the primary key. There are no transitive dependencies.

To be in BCNF, a table must meet the following requirement:

- In all the tables, every determinant is a candidate key, meaning that no non-key attribute is dependent on another non-superkey attribute. Therefore, all the tables are in BCNF.

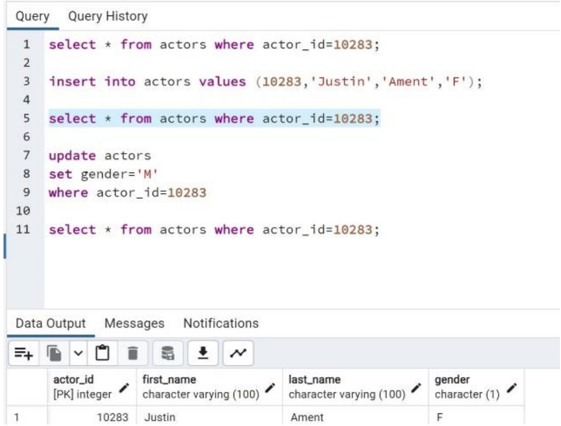
VII. QUERIES



```
select * from actors where actor_id=10283;

insert into actors values (10283,'Justin','Ament','F');

select * from actors where actor_id=10283;

update actors
set gender='M'
where actor_id=10283

select * from actors where actor_id=10283;
```

	actor_id [PK] integer	first_name character varying (100)	last_name character varying (100)	gender character (1)

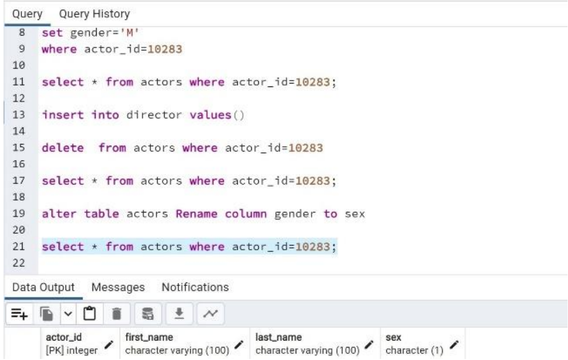
1. Insert



```
select * from actors where actor_id=10283;

insert into actors values (10283,'Justin','Ament','F');

select * from actors where actor_id=10283;

update actors
set gender='M'
where actor_id=10283

select * from actors where actor_id=10283;
```

	actor_id [PK] integer	first_name character varying (100)	last_name character varying (100)	gender character (1)
1	10283	Justin	Ament	F

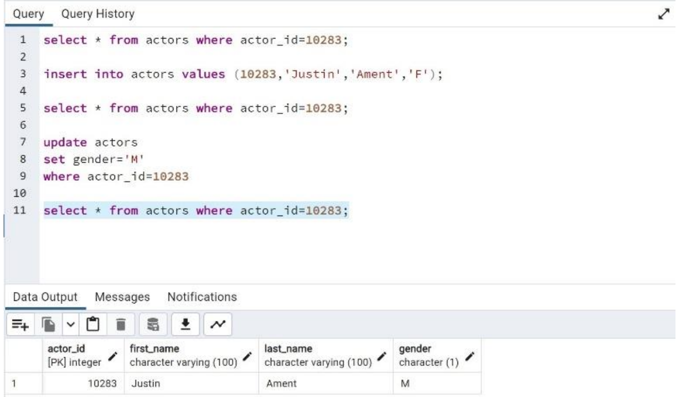
2. Alter



```
set gender='M'
where actor_id=10283

select * from actors where actor_id=10283;

insert into director values()

delete  from actors where actor_id=10283

select * from actors where actor_id=10283;

alter table actors Rename column gender to sex

select * from actors where actor_id=10283;

```

	actor_id [PK] integer	first_name character varying (100)	last_name character varying (100)	sex character (1)

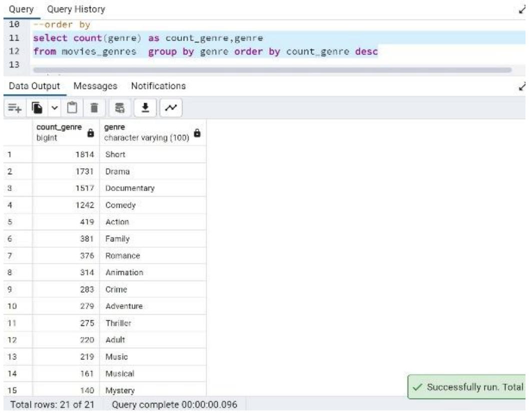
3. Update



```
select * from actors where actor_id=10283;

insert into actors values (10283,'Justin','Ament','F');

select * from actors where actor_id=10283;

update actors
set gender='M'
where actor_id=10283

select * from actors where actor_id=10283;
```

	actor_id [PK] integer	first_name character varying (100)	last_name character varying (100)	gender character (1)
1	10283	Justin	Ament	M

4. Order By



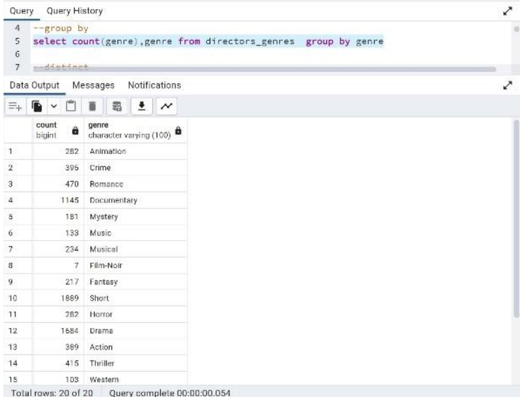
```
--order by
select count(genre) as count_genre,genre
from movies_genres  group by genre order by count_genre desc

```

	count_genre bigint	genre character varying (100)
1	1814	Short
2	1731	Drama
3	1517	Documentary
4	1242	Comedy
5	419	Action
6	381	Family
7	376	Romance
8	314	Animation
9	283	Crime
10	279	Adventure
11	275	Thriller
12	220	Adult
13	219	Music
14	161	Musical
15	140	Mystery

✓ Successfully run. Total

Total rows: 21 of 21   Query complete 00:00:00.096

5. Group By

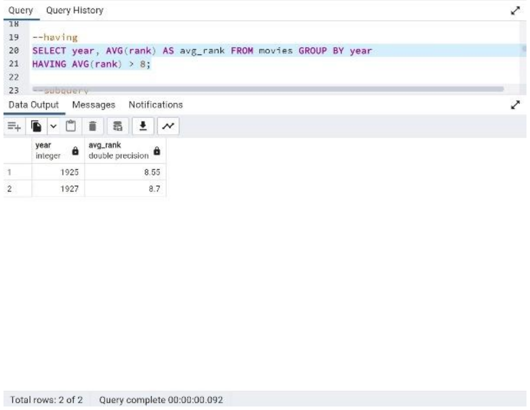


```sql
--group by
select count(genre),genre from directors_genres  group by genre

--distinct
```

	count bigint	genre character varying (100)
1	282	Animation
2	395	Crime
3	470	Romance
4	1145	Documentary
5	181	Mystery
6	133	Music
7	234	Musical
8	7	Film-Noir
9	217	Fantasy
10	1889	Short
11	282	Horror
12	1684	Drama
13	389	Action
14	415	Thriller
15	103	Western

Total rows: 20 of 20   Query complete 00:00:00.054

6. Having

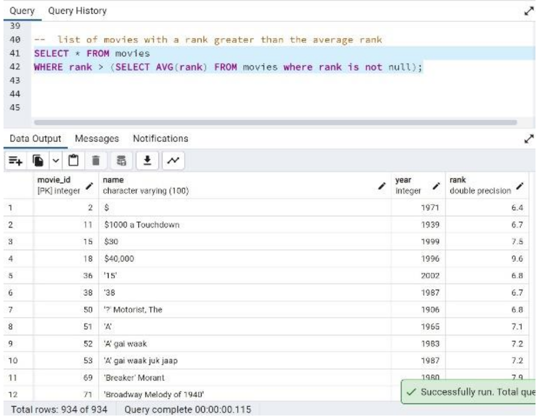


```sql
--having
SELECT year, AVG(rank) AS avg_rank FROM movies GROUP BY year
HAVING AVG(rank) > 8;

--subquery
```

	year integer	avg_rank double precision
1	1925	8.55
2	1927	8.7

Total rows: 2 of 2   Query complete 00:00:00.092

7. List of movies with rank greater than average rank



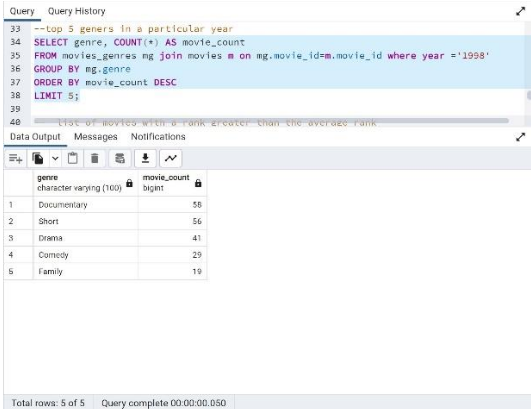
```sql
--  list of movies with a rank greater than the average rank
SELECT * FROM movies
WHERE rank > (SELECT AVG(rank) FROM movies where rank is not null);
```

	movie_id [PK] integer	name character varying (100)	year integer	rank double precision
1	2	$	1971	6.4
2	11	$1000 a Touchdown	1939	6.7
3	15	$30	1999	7.5
4	18	$40,000	1996	9.6
5	36	'15'	2002	6.8
6	38	'38	1987	6.7
7	50	'?' Motorist, The	1906	6.8
8	51	'A'	1965	7.1
9	52	'A' gai waak	1983	7.2
10	53	'A' gai waak juk jaap	1987	7.2
11	69	'Breaker' Morant	1980	7.9
12	71	'Broadway Melody of 1940'		

✓ Successfully run. Total que

Total rows: 934 of 934   Query complete 00:00:00.115

8. Top 5 genres in a particular year

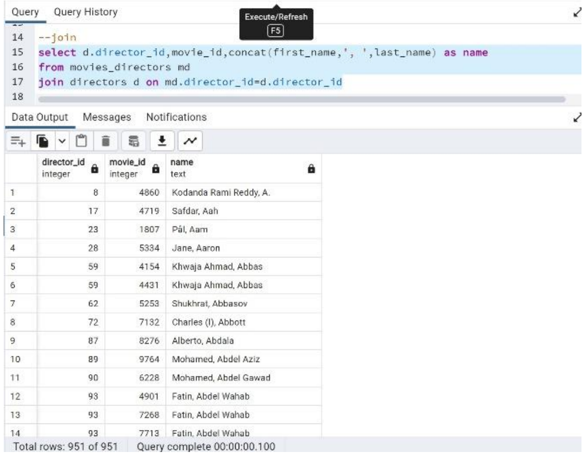


```sql
--top 5 geners in a particular year
SELECT genre, COUNT(*) AS movie_count
FROM movies_genres mg join movies m on mg.movie_id=m.movie_id where year ='1998'
GROUP BY mg.genre
ORDER BY movie_count DESC
LIMIT 5;

--   list of movies with a rank greater than the average rank
```

	genre character varying (100)	movie_count bigint
1	Documentary	58
2	Short	56
3	Drama	41
4	Comedy	29
5	Family	19

Total rows: 5 of 5   Query complete 00:00:00.050

9. Join

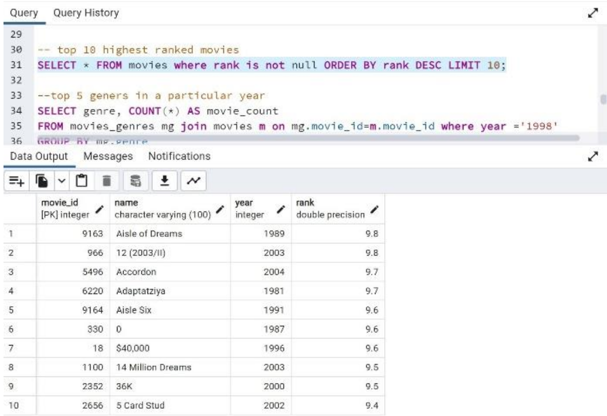


```sql
--join
select d.director_id,movie_id,concat(first_name,', ',last_name) as name
from movies_directors md
join directors d on md.director_id=d.director_id
```

	director_id integer	movie_id integer	name text
1	8	4860	Kodanda Rami Reddy, A.
2	17	4719	Safdar, Aah
3	23	1807	Pál, Aam
4	28	5334	Jane, Aaron
5	59	4154	Khwaja Ahmad, Abbas
6	59	4431	Khwaja Ahmad, Abbas
7	62	5253	Shukhrat, Abbasov
8	72	7132	Charles (I), Abbott
9	87	8276	Alberto, Abdala
10	89	9764	Mohamed, Abdel Aziz
11	90	6228	Mohamed, Abdel Gawad
12	93	4901	Fatin, Abdel Wahab
13	93	7268	Fatin, Abdel Wahab
14	93	7713	Fatin, Abdel Wahab

Total rows: 951 of 951   Query complete 00:00:00.100

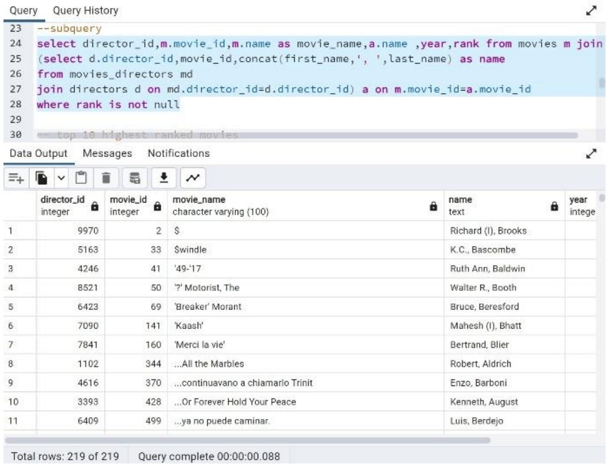
10. Top 10 highest ranked movies



```sql
-- top 10 highest ranked movies
SELECT * FROM movies where rank is not null ORDER BY rank DESC LIMIT 10;

--top 5 geners in a particular year
SELECT genre, COUNT(*) AS movie_count
FROM movies_genres mg join movies m on mg.movie_id=m.movie_id where year ='1998'
GROUP BY mg.genre
```

	movie_id [PK] integer	name character varying (100)	year integer	rank double precision
1	9163	Aisle of Dreams	1989	9.8
2	966	12 (2003/II)	2003	9.8
3	5496	Accordon	2004	9.7
4	6220	Adaptatziya	1981	9.7
5	9164	Aisle Six	1991	9.6
6	330	0	1987	9.6
7	18	$40,000	1996	9.6
8	1100	14 Million Dreams	2003	9.5
9	2352	36K	2000	9.5
10	2656	5 Card Stud	2002	9.4

11. SubQuery



```sql
--subquery
select director_id,m.movie_id,m.name as movie_name,a.name ,year,rank from movies m join
(select d.director_id,movie_id,concat(first_name,', ',last_name) as name
from movies_directors md
join directors d on md.director_id=d.director_id) a on m.movie_id=a.movie_id
where rank is not null

-- top 10 highest ranked movies
```

	director_id integer	movie_id integer	movie_name character varying (100)	name text	year intege
1	9970	2	$	Richard (I), Brooks	
2	5163	33	$windle	K.C., Bascombe	
3	4246	41	'49-'17	Ruth Ann, Baldwin	
4	8521	50	'?' Motorist, The	Walter R., Booth	
5	6423	69	'Breaker' Morant	Bruce, Beresford	
6	7090	141	'Kaash'	Mahesh (I), Bhatt	
7	7841	160	'Merci la vie'	Bertrand, Blier	
8	1102	344	...All the Marbles	Robert, Aldrich	
9	4616	370	...continuavano a chiamarlo Trinit	Enzo, Barboni	
10	3393	428	...Or Forever Hold Your Peace	Kenneth, August	
11	6409	499	...ya no puede caminar.	Luis, Berdejo	

Total rows: 219 of 219   Query complete 00:00:00.088

VIII. QUERY EXECUTION ANALYSIS

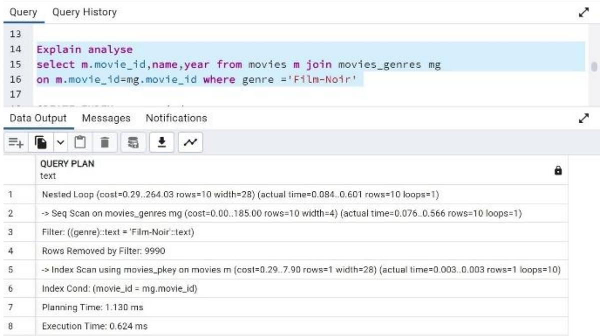
EXPLAIN ANALYZE is a tool in database management systems that provides information on how a query is executed by the database engine. It is a combination of two sql commands, Explain and Analyze, and is commonly used for query optimization and performance tuning.

INDEXING:

Indexing is a database optimization technique that is used to improve the speed and efficiency of queries by reducing the time it takes to search for data. It involves creating a separate data structure that maps the values in one or more columns of a table to their physical location on disk.

Query-1:

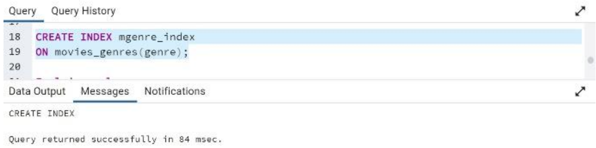
Before Indexing:



```
Explain analyse
select m.movie_id,name,year from movies m join movies_genres mg
on m.movie_id=mg.movie_id where genre ='Film-Noir'
```

	QUERY PLAN text
1	Nested Loop (cost=0.29..264.03 rows=10 width=28) (actual time=0.084..0.601 rows=10 loops=1)
2	-> Seq Scan on movies_genres mg (cost=0.00..185.00 rows=10 width=4) (actual time=0.076..0.566 rows=10 loops=1)
3	Filter: ((genre)::text = 'Film-Noir'::text)
4	Rows Removed by Filter: 9990
5	-> Index Scan using movies_pkey on movies m (cost=0.29..7.90 rows=1 width=28) (actual time=0.003..0.003 rows=1 loops=10)
6	Index Cond: (movie_id = mg.movie_id)
7	Planning Time: 1.130 ms
8	Execution Time: 0.624 ms

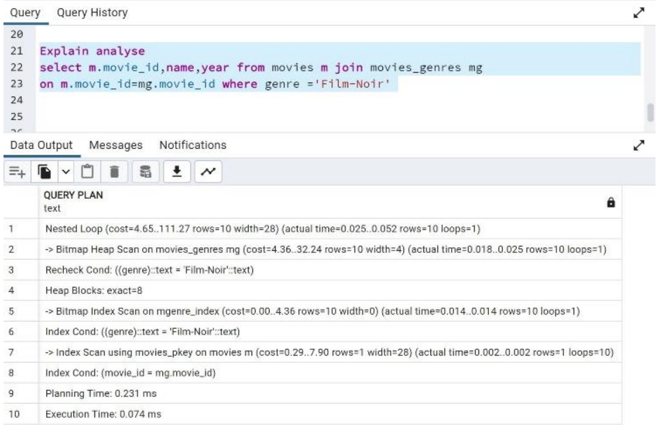
Indexing:



```
CREATE INDEX mgenre_index
ON movies_genres(genre);
```

```
CREATE INDEX

Query returned successfully in 84 msec.
```

After Indexing:

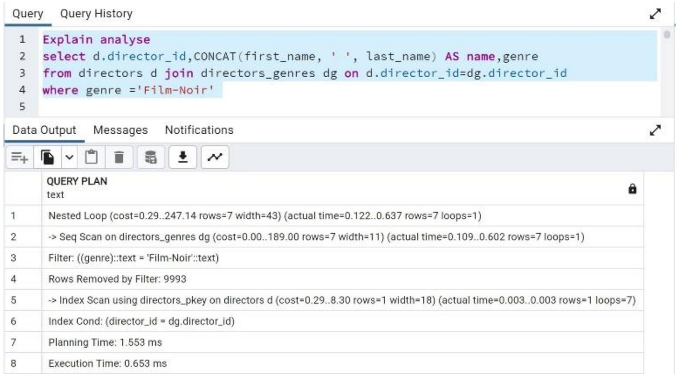


```
Explain analyse
select m.movie_id,name,year from movies m join movies_genres mg
on m.movie_id=mg.movie_id where genre ='Film-Noir'
```

	QUERY PLAN text
1	Nested Loop (cost=4.65..111.27 rows=10 width=28) (actual time=0.025..0.052 rows=10 loops=1)
2	-> Bitmap Heap Scan on movies_genres mg (cost=4.36..32.24 rows=10 width=4) (actual time=0.018..0.025 rows=10 loops=1)
3	Recheck Cond: ((genre)::text = 'Film-Noir'::text)
4	Heap Blocks: exact=8
5	-> Bitmap Index Scan on mgenre_index (cost=0.00..4.36 rows=10 width=0) (actual time=0.014..0.014 rows=10 loops=1)
6	Index Cond: ((genre)::text = 'Film-Noir'::text)
7	-> Index Scan using movies_pkey on movies m (cost=0.29..7.90 rows=1 width=28) (actual time=0.002..0.002 rows=1 loops=10)
8	Index Cond: (movie_id = mg.movie_id)
9	Planning Time: 0.231 ms
10	Execution Time: 0.074 ms

Query-2:

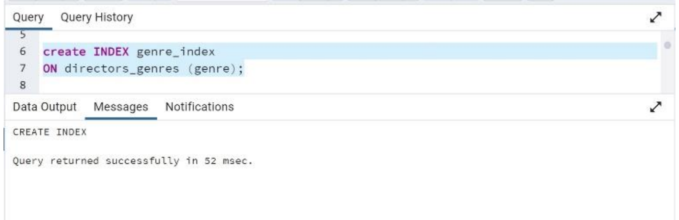
Before Indexing:



```
Explain analyse
select d.director_id,CONCAT(first_name, ' ', last_name) AS name,genre
from directors d join directors_genres dg on d.director_id=dg.director_id
where genre ='Film-Noir'
```

	QUERY PLAN text
1	Nested Loop (cost=0.29..247.14 rows=7 width=43) (actual time=0.122..0.637 rows=7 loops=1)
2	-> Seq Scan on directors_genres dg (cost=0.00..189.00 rows=7 width=11) (actual time=0.109..0.602 rows=7 loops=1)
3	Filter: ((genre)::text = 'Film-Noir'::text)
4	Rows Removed by Filter: 9993
5	-> Index Scan using directors_pkey on directors d (cost=0.29..8.30 rows=1 width=18) (actual time=0.003..0.003 rows=1 loops=7)
6	Index Cond: (director_id = dg.director_id)
7	Planning Time: 1.553 ms
8	Execution Time: 0.653 ms

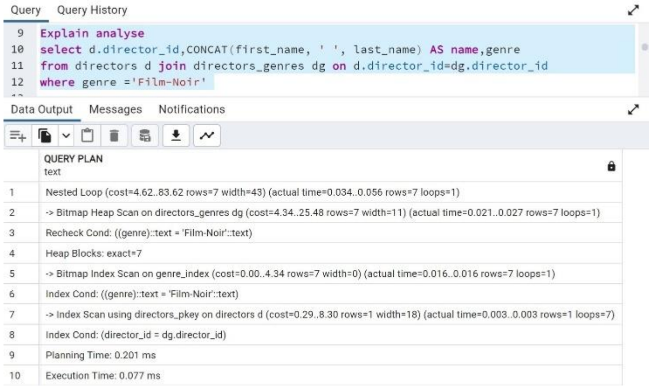
Indexing:



```
create INDEX genre_index
ON directors_genres (genre);
```

```
CREATE INDEX

Query returned successfully in 52 msec.
```

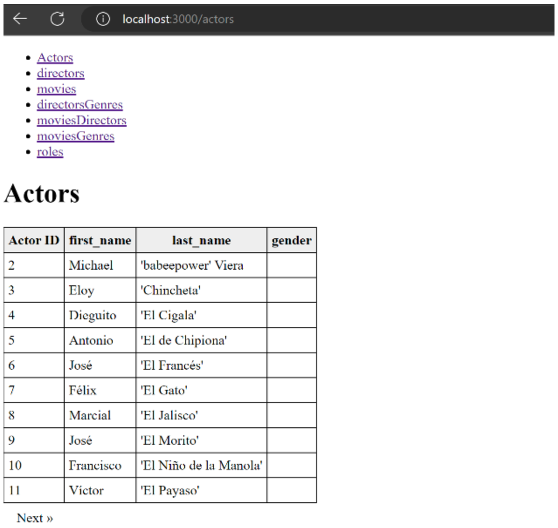
After Indexing:



```
Explain analyse
select d.director_id,CONCAT(first_name, ' ', last_name) AS name,genre
from directors d join directors_genres dg on d.director_id=dg.director_id
where genre ='Film-Noir'
```

	QUERY PLAN text
1	Nested Loop (cost=4.62..83.62 rows=7 width=43) (actual time=0.034..0.056 rows=7 loops=1)
2	-> Bitmap Heap Scan on directors_genres dg (cost=4.34..25.48 rows=7 width=11) (actual time=0.021..0.027 rows=7 loops=1)
3	Recheck Cond: ((genre)::text = 'Film-Noir'::text)
4	Heap Blocks: exact=7
5	-> Bitmap Index Scan on genre_index (cost=0.00..4.34 rows=7 width=0) (actual time=0.016..0.016 rows=7 loops=1)
6	Index Cond: ((genre)::text = 'Film-Noir'::text)
7	-> Index Scan using directors_pkey on directors d (cost=0.29..8.30 rows=1 width=18) (actual time=0.003..0.003 rows=1 loops=7)
8	Index Cond: (director_id = dg.director_id)
9	Planning Time: 0.201 ms
10	Execution Time: 0.077 ms

IX. WEBSITE

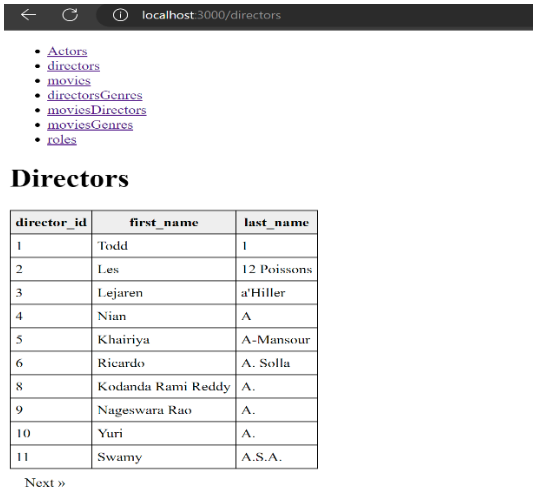


- Actors
- directors
- movies
- directorsGenres
- moviesDirectors
- moviesGenres
- roles

Actors

Actor ID	first_name	last_name	gender
2	Michael	'babeepower' Viera	
3	Eloy	'Chincheta'	
4	Dieguito	'El Cigala'	
5	Antonio	'El de Chipiona'	
6	José	'El Francés'	
7	Félix	'El Gato'	
8	Marcial	'El Jalisco'	
9	José	'El Morito'	
10	Francisco	'El Niño de la Manola'	
11	Victor	'El Payaso'	

Next »

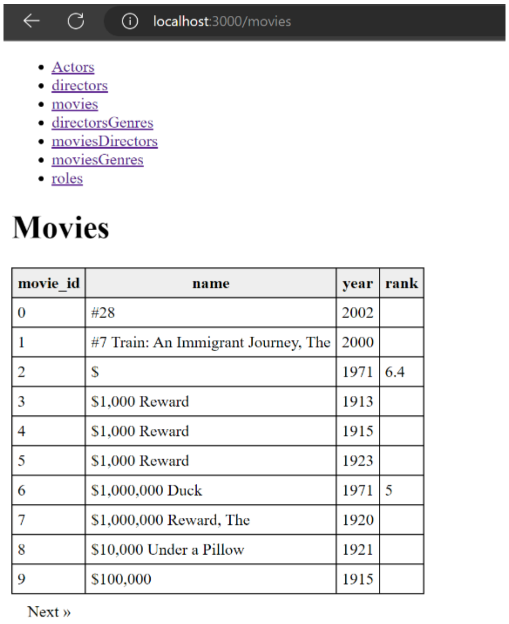


- Actors
- directors
- movies
- directorsGenres
- moviesDirectors
- moviesGenres
- roles

Directors

director_id	first_name	last_name
1	Todd	1
2	Les	12 Poissons
3	Lejaren	a'Hiller
4	Nian	A
5	Khairiya	A-Mansour
6	Ricardo	A. Solla
8	Kodanda Rami Reddy	A.
9	Nageswara Rao	A.
10	Yuri	A.
11	Swamy	A.S.A.

Next »

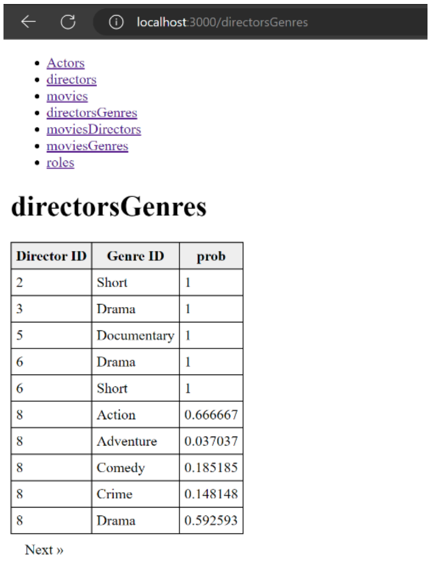


- Actors
- directors
- movies
- directorsGenres
- moviesDirectors
- moviesGenres
- roles

Movies

movie_id	name	year	rank
0	#28	2002	
1	#7 Train: An Immigrant Journey, The	2000	
2	$	1971	6.4
3	$1,000 Reward	1913	
4	$1,000 Reward	1915	
5	$1,000 Reward	1923	
6	$1,000,000 Duck	1971	5
7	$1,000,000 Reward, The	1920	
8	$10,000 Under a Pillow	1921	
9	$100,000	1915	

Next »

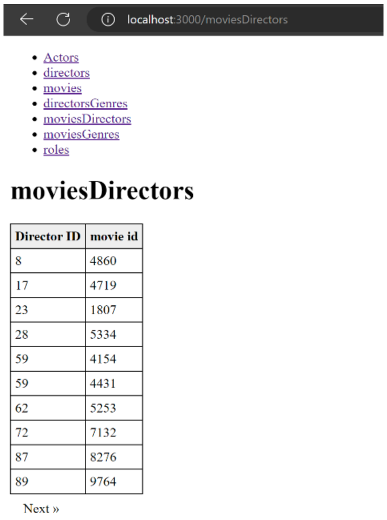


- Actors
- directors
- movies
- directorsGenres
- moviesDirectors
- moviesGenres
- roles

directorsGenres

Director ID	Genre ID	prob
2	Short	1
3	Drama	1
5	Documentary	1
6	Drama	1
6	Short	1
8	Action	0.666667
8	Adventure	0.037037
8	Comedy	0.185185
8	Crime	0.148148
8	Drama	0.592593

Next »

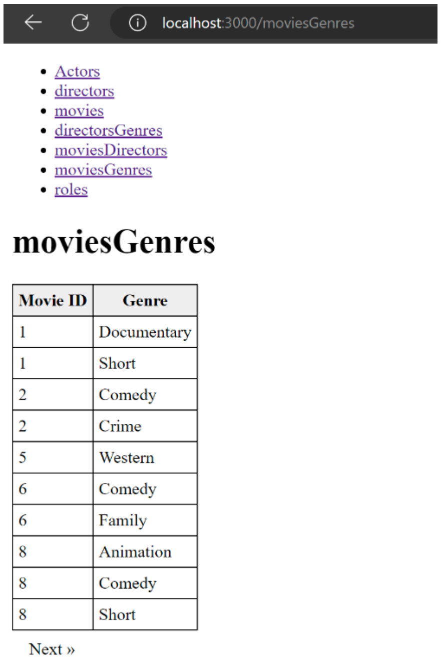


- Actors
- directors
- movies
- directorsGenres
- moviesDirectors
- moviesGenres
- roles

moviesDirectors

Director ID	movie id
8	4860
17	4719
23	1807
28	5334
59	4154
59	4431
62	5253
72	7132
87	8276
89	9764

Next »

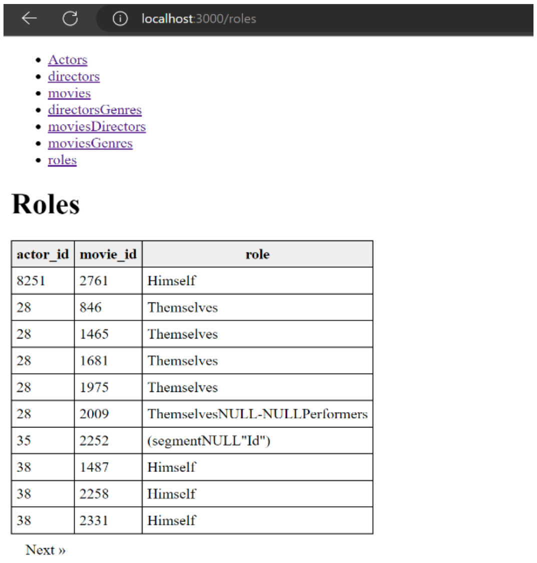


- Actors
- directors
- movies
- directorsGenres
- moviesDirectors
- moviesGenres
- roles

moviesGenres

Movie ID	Genre
1	Documentary
1	Short
2	Comedy
2	Crime
5	Western
6	Comedy
6	Family
8	Animation
8	Comedy
8	Short

Next »



- Actors
- directors
- movies
- directorsGenres
- moviesDirectors
- moviesGenres
- roles

Roles

actor_id	movie_id	role
8251	2761	Himself
28	846	Themselves
28	1465	Themselves
28	1681	Themselves
28	1975	Themselves
28	2009	ThemselvesNULL-NULLPerformers
35	2252	(segmentNULL"Id")
38	1487	Himself
38	2258	Himself
38	2331	Himself

Next »

X. REFERENCES

[1] https://sqlzoo.net/wiki/SQL_Tutorial

[2] https://www.mysql.com/products/workbench/

[3] https://www.w3schools.com/sql/default.asp

[4] https://relational.fit.cvut.cz/dataset/IMDb

[5] https://learn.microsoft.com/en-us/office/troubleshoot/access/database-normalization-description

[6] https://www.mongodb.com/docs/manual/indexes/