DATABASE PROJECT PART-4

By: Pratyusha Sanapathi

---

1. New questions:

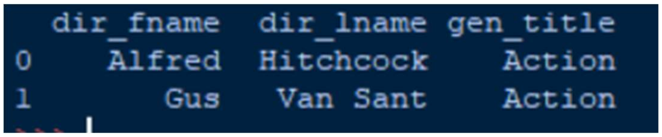
Queries:

1. **Write a SQL query to find the directors who have directed films in Action genre. Group the result set on director first name, last name and generic title. Sort the result-set in ascending order by director first name and last name. Return director first name, last name and number of genres movies.**

ANS:

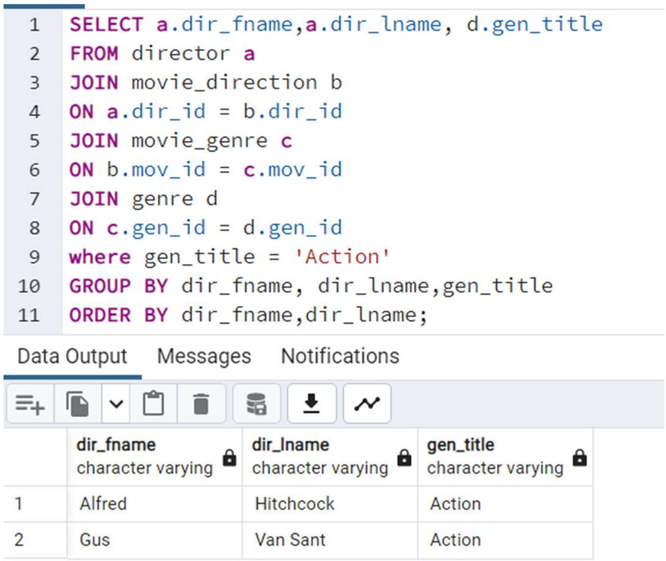
```
SELECT a.dir_fname,a.dir_lname, d.gen_title
FROM director a
JOIN movie_direction b
ON a.dir_id = b.dir_id
JOIN movie_genre c
ON b.mov_id = c.mov_id
JOIN genre d
ON c.gen_id = d.gen_id
where gen_title = 'Action'
GROUP BY dir_fname, dir_lname,gen_title
ORDER BY dir_fname,dir_lname;
```

<u>Output using Python:</u>



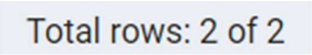
```
  dir_fname  dir_lname gen_title
0    Alfred  Hitchcock    Action
1       Gus   Van Sant    Action
```

<u>Output using pgAdmin:</u>



```
SELECT a.dir_fname,a.dir_lname, d.gen_title
FROM director a
JOIN movie_direction b
ON a.dir_id = b.dir_id
JOIN movie_genre c
ON b.mov_id = c.mov_id
JOIN genre d
ON c.gen_id = d.gen_id
where gen_title = 'Action'
GROUP BY dir_fname, dir_lname,gen_title
ORDER BY dir_fname,dir_lname;
```

	dir_fname character varying	dir_lname character varying	gen_title character varying
1	Alfred	Hitchcock	Action
2	Gus	Van Sant	Action



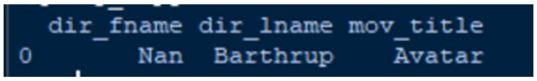
Total rows: 2 of 2

2. **Write a SQL query to find the director who directed a movie that featured a role in 'Avatar'. Return director first name, last name and movie title.**

ANS:

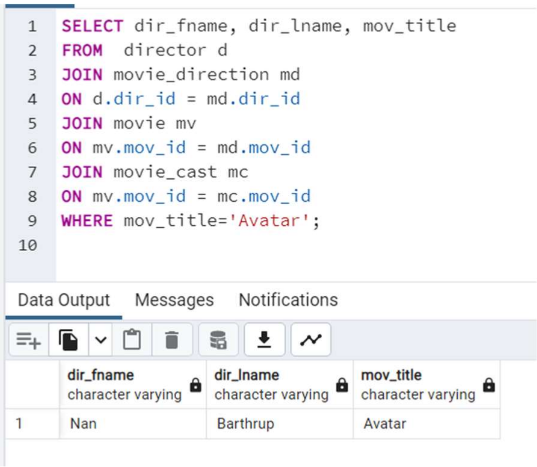
SELECT dir_fname, dir_lname, mov_title
FROM director d
JOIN movie_direction md
ON d.dir_id = md.dir_id
JOIN movie mv
ON mv.mov_id = md.mov_id
JOIN movie_cast mc
ON mv.mov_id = mc.mov_id
WHERE mov_title='Avatar';

<u>Output using Python:</u>



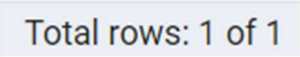
```
  dir_fname dir_lname mov_title
0       Nan  Barthrup    Avatar
```

<u>Output using pgAdmin:</u>



```sql
SELECT dir_fname, dir_lname, mov_title
FROM  director d
JOIN movie_direction md
ON d.dir_id = md.dir_id
JOIN movie mv
ON mv.mov_id = md.mov_id
JOIN movie_cast mc
ON mv.mov_id = mc.mov_id
WHERE mov_title='Avatar';
```

	dir_fname character varying	dir_lname character varying	mov_title character varying
1	Nan	Barthrup	Avatar

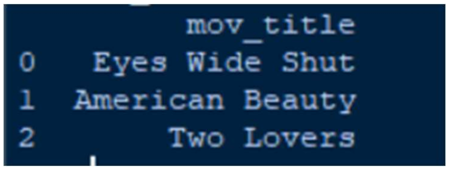


Total rows: 1 of 1

3. **write a SQL query to find the movie that was released in 1999. Return movie title.**

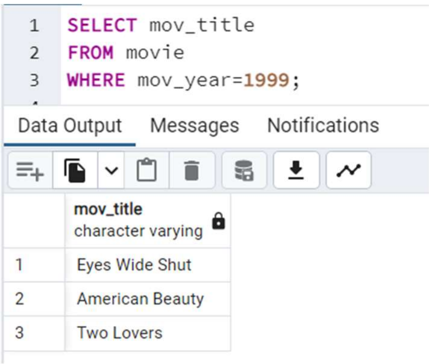
ANS: SELECT mov_title
FROM movie
WHERE mov_year=1999;

<u>Output using Python:</u>



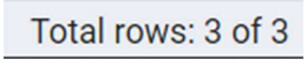
```
         mov_title
0   Eyes Wide Shut
1  American Beauty
2      Two Lovers
```

<u>Output using pgAdmin:</u>



```sql
SELECT mov_title
FROM movie
WHERE mov_year=1999;
```

	mov_title character varying
1	Eyes Wide Shut
2	American Beauty
3	Two Lovers



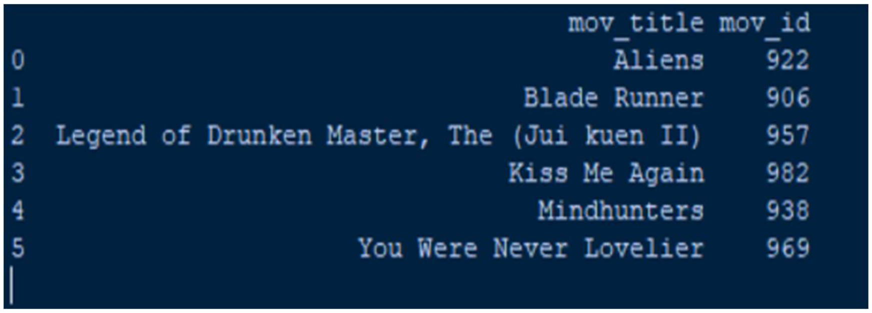
Total rows: 3 of 3

4. **write a SQL query to search for movies that do not have any ratings. Return movie title.**

ANS:

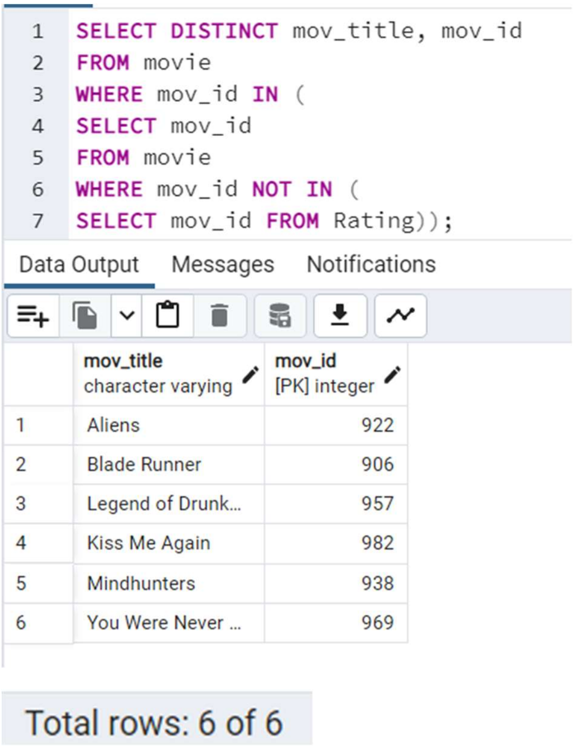
```
SELECT DISTINCT mov_title, mov_id
FROM movie
WHERE mov_id IN (
SELECT mov_id
FROM movie
WHERE mov_id NOT IN (
SELECT mov_id FROM Rating));
```

Output using Python:



```
                                  mov_title mov_id
0                                    Aliens    922
1                              Blade Runner    906
2  Legend of Drunken Master, The (Jui kuen II)    957
3                             Kiss Me Again    982
4                               Mindhunters    938
5                    You Were Never Lovelier    969
```

Output using pgAdmin:



```
SELECT DISTINCT mov_title, mov_id
FROM movie
WHERE mov_id IN (
SELECT mov_id
FROM movie
WHERE mov_id NOT IN (
SELECT mov_id FROM Rating));
```

	mov_title character varying	mov_id [PK] integer
1	Aliens	922
2	Blade Runner	906
3	Legend of Drunk...	957
4	Kiss Me Again	982
5	Mindhunters	938
6	You Were Never ...	969

Total rows: 6 of 6

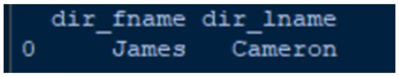
5. **write a SQL query to find the director of a film that cast a role in 'Titanic'. Return director first name, last name.**

ANS:

```
SELECT dir_fname, dir_lname
FROM  director
WHERE dir_id in (SELECT dir_id
                 FROM movie_direction
```

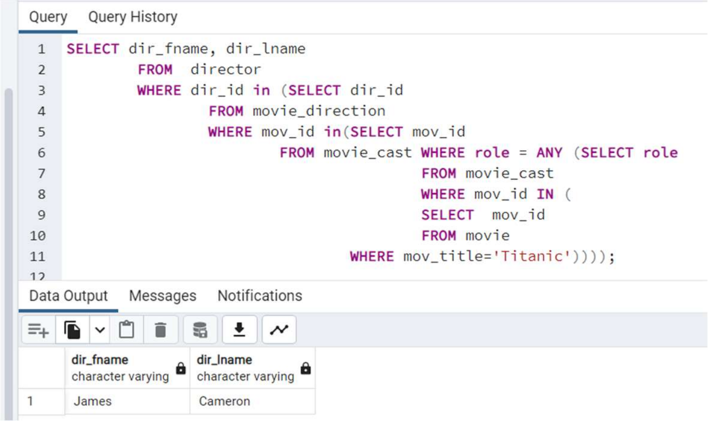
```
                                   WHERE mov_id in(SELECT mov_id
                                             FROM movie_cast WHERE role = ANY (SELECT
role
                                                                      FROM movie_cast
                                                                      WHERE mov_id IN
(
                                                                      SELECT  mov_id
                                                                      FROM movie
                                                           WHERE mov_title='Titanic'))));
```

<u>Output using Python:</u>



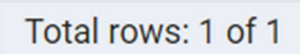
```
  dir_fname dir_lname
0     James   Cameron
```

<u>Output using pgAdmin:</u>



```
SELECT dir_fname, dir_lname
        FROM  director
        WHERE dir_id in (SELECT dir_id
                FROM movie_direction
                WHERE mov_id in(SELECT mov_id
                        FROM movie_cast WHERE role = ANY (SELECT role
                                FROM movie_cast
                                WHERE mov_id IN (
                                SELECT  mov_id
                                FROM movie
                            WHERE mov_title='Titanic'))));
```

	dir_fname character varying	dir_lname character varying
1	James	Cameron

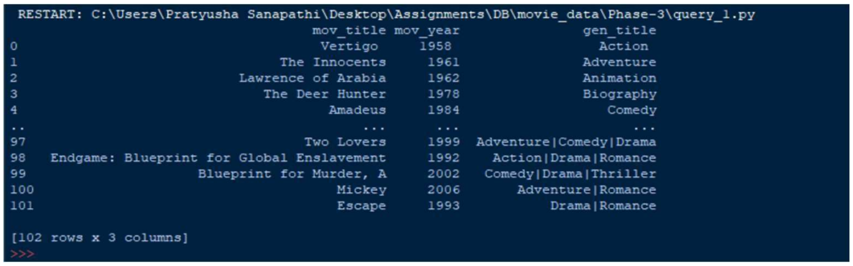


Total rows: 1 of 1

6. write a SQL query to find the movies with year and genres. Return movie title, movie year and generic title.

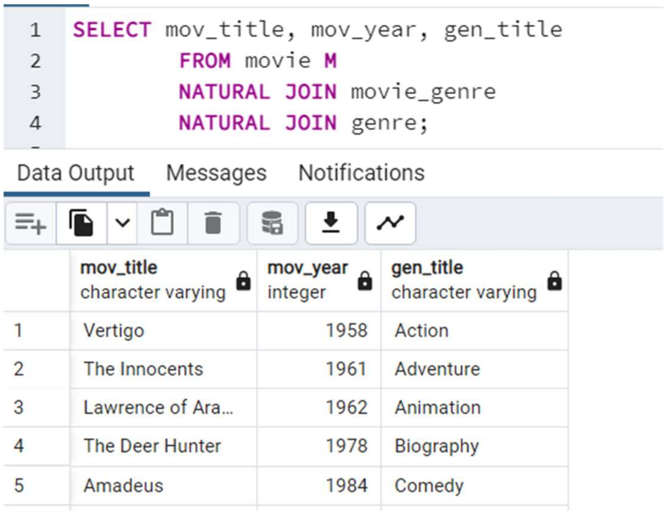
ANS: SELECT mov_title, mov_year, gen_title
FROM movie M
NATURAL JOIN movie_genres
NATURAL JOIN genres;

<u>Output using Python:</u>



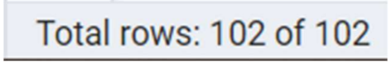
```
 RESTART: C:\Users\Pratyusha Sanapathi\Desktop\Assignments\DB\movie_data\Phase-3\query_1.py
                                   mov_title mov_year                gen_title
0                                    Vertigo     1958                   Action
1                              The Innocents     1961                Adventure
2                          Lawrence of Arabia     1962                Animation
3                             The Deer Hunter     1978                Biography
4                                    Amadeus     1984                   Comedy
..                                       ...      ...                      ...
97                                 Two Lovers     1999  Adventure|Comedy|Drama
98   Endgame: Blueprint for Global Enslavement     1992    Action|Drama|Romance
99                       Blueprint for Murder, A     2002   Comedy|Drama|Thriller
100                                    Mickey     2006       Adventure|Romance
101                                    Escape     1993           Drama|Romance

[102 rows x 3 columns]
>>>
```

Output using pgAdmin:



```
SELECT mov_title, mov_year, gen_title
        FROM movie M
        NATURAL JOIN movie_genre
        NATURAL JOIN genre;
```

	mov_title character varying	mov_year integer	gen_title character varying
1	Vertigo	1958	Action
2	The Innocents	1961	Adventure
3	Lawrence of Ara...	1962	Animation
4	The Deer Hunter	1978	Biography
5	Amadeus	1984	Comedy

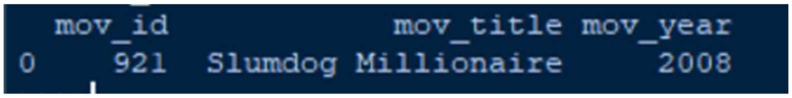


Total rows: 102 of 102

7. **write a SQL query to find the movie titles that starts with the word 'Slumdog'. Sort the result order by movie year. Return movie ID, movie title and movie release year.**

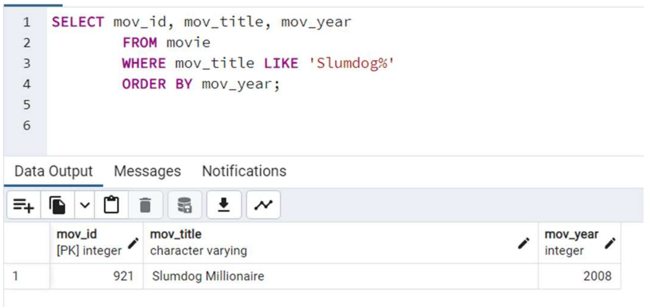
ANS: SELECT mov_id, mov_title, mov_year
FROM movie
WHERE mov_title LIKE 'Slumdog%'
ORDER BY mov_year;

Output using Python:



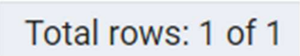
```
   mov_id          mov_title mov_year
0     921  Slumdog Millionaire     2008
```

Output using pgAdmin:



```
SELECT mov_id, mov_title, mov_year
        FROM movie
        WHERE mov_title LIKE 'Slumdog%'
        ORDER BY mov_year;
```

	mov_id [PK] integer	mov_title character varying	mov_year integer
1	921	Slumdog Millionaire	2008



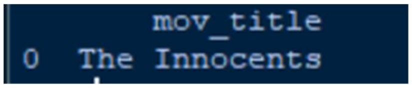
Total rows: 1 of 1

8. **write a SQL query to find the movies directed by 'James Cameron'. Return movie title.**

ANS:

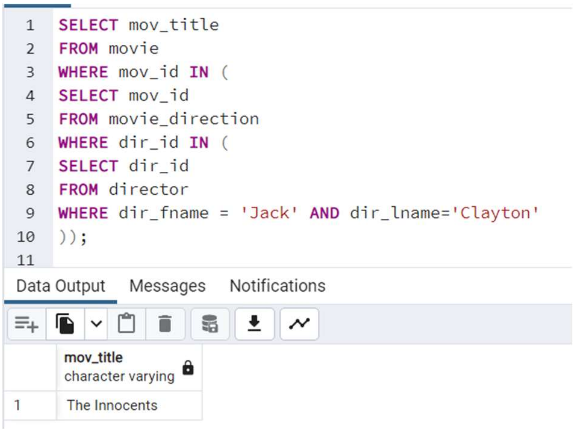
SELECT mov_title
FROM movie
WHERE mov_id IN (
SELECT mov_id
FROM movie_direction
WHERE dir_id IN (
SELECT dir_id
FROM director
WHERE dir_fname = 'Jack' AND dir_lname='Clayton'
));

<u>Output using Python:</u>



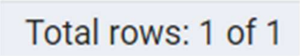
```
        mov_title
0   The Innocents
```

<u>Output using pgAdmin:</u>



```
SELECT mov_title
FROM movie
WHERE mov_id IN (
SELECT mov_id
FROM movie_direction
WHERE dir_id IN (
SELECT dir_id
FROM director
WHERE dir_fname = 'Jack' AND dir_lname='Clayton'
));

```

	mov_title character varying
1	The Innocents



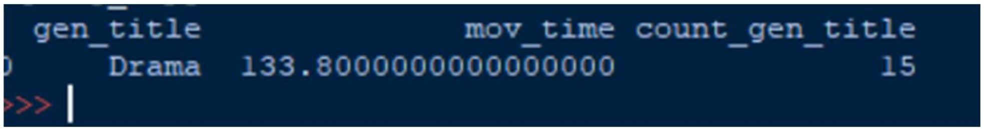
Total rows: 1 of 1

9. **write a SQL query to calculate the average movie length and count the number of movies in each genre. Return genre title, average time and number of movies for each genre.**

ANS:

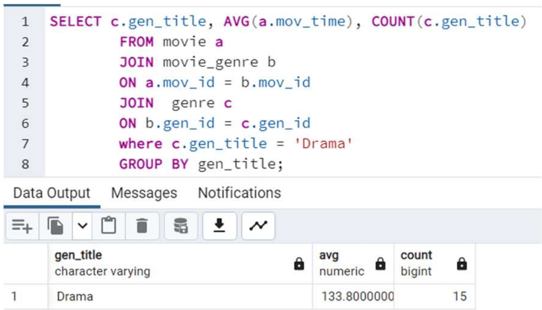
SELECT c.gen_title, AVG(a.mov_time), COUNT(c.gen_title)
FROM movie a
JOIN movie_genre b
ON a.mov_id = b.mov_id
JOIN genre c
ON b.gen_id = c.gen_id
where c.gen_title = 'Drama'
GROUP BY gen_title;

<u>Output using Python:</u>



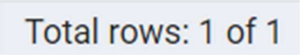
```
  gen_title                  mov_time  count_gen_title
0     Drama  133.8000000000000000                   15
>>> |
```

Output using pgAdmin:



```
SELECT c.gen_title, AVG(a.mov_time), COUNT(c.gen_title)
        FROM movie a
        JOIN movie_genre b
        ON a.mov_id = b.mov_id
        JOIN  genre c
        ON b.gen_id = c.gen_id
        where c.gen_title = 'Drama'
        GROUP BY gen_title;
```

	gen_title character varying	avg numeric	count bigint
1	Drama	133.8000000	15



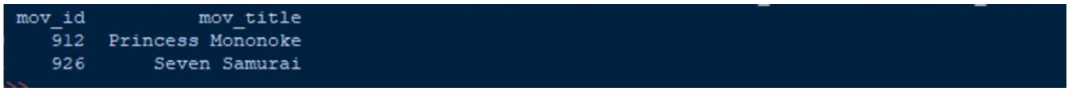
Total rows: 1 of 1

10. write a SQL query to find those movies, which were released before 1998 and language is Japanese.

ANS:

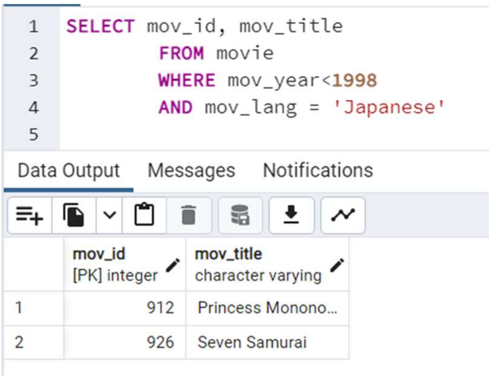
```
SELECT mov_id, mov_title
FROM movie
WHERE mov_year<1998
AND mov_lang = 'Japanese'
```

Output using Python:



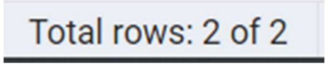
```
mov_id          mov_title
   912  Princess Mononoke
   926      Seven Samurai
```

Output using pgAdmin:



```
SELECT mov_id, mov_title
        FROM movie
        WHERE mov_year<1998
        AND mov_lang = 'Japanese'
```

	mov_id [PK] integer	mov_title character varying
1	912	Princess Monono...
2	926	Seven Samurai



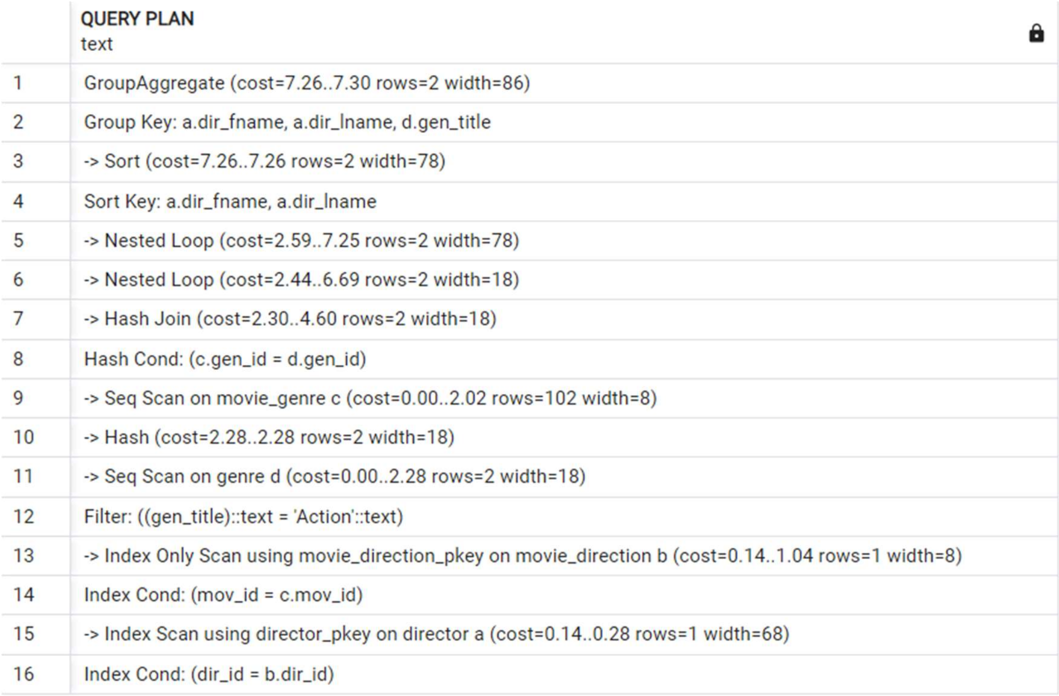
Total rows: 2 of 2

2. Query Performance:

Query: (Q1 in section 1)

```
SELECT a.dir_fname,a.dir_lname, d.gen_title,count(d.gen_title)
 FROM director a
 JOIN movie_direction b
 ON a.dir_id = b.dir_id
 JOIN movie_genre c
 ON b.mov_id = c.mov_id
 JOIN genre d
 ON c.gen_id = d.gen_id
  where gen_title = 'Action'
 GROUP BY dir_fname, dir_lname,gen_title
 ORDER BY dir_fname,dir_lname;
```

Output of EXPLAIN command:



	QUERY PLAN text
1	GroupAggregate (cost=7.26..7.30 rows=2 width=86)
2	Group Key: a.dir_fname, a.dir_lname, d.gen_title
3	-> Sort (cost=7.26..7.26 rows=2 width=78)
4	Sort Key: a.dir_fname, a.dir_lname
5	-> Nested Loop (cost=2.59..7.25 rows=2 width=78)
6	-> Nested Loop (cost=2.44..6.69 rows=2 width=18)
7	-> Hash Join (cost=2.30..4.60 rows=2 width=18)
8	Hash Cond: (c.gen_id = d.gen_id)
9	-> Seq Scan on movie_genre c (cost=0.00..2.02 rows=102 width=8)
10	-> Hash (cost=2.28..2.28 rows=2 width=18)
11	-> Seq Scan on genre d (cost=0.00..2.28 rows=2 width=18)
12	Filter: ((gen_title)::text = 'Action'::text)
13	-> Index Only Scan using movie_direction_pkey on movie_direction b (cost=0.14..1.04 rows=1 width=8)
14	Index Cond: (mov_id = c.mov_id)
15	-> Index Scan using director_pkey on director a (cost=0.14..0.28 rows=1 width=68)
16	Index Cond: (dir_id = b.dir_id)

Join Algorithm Used: Nested Loop, Hash Join

Reason:

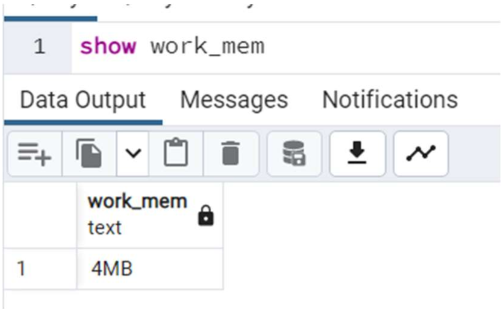
Postgres uses Hash join for the condition on ‘movie_genre’ and ‘genre’ tables.

Postgres also uses two Nested loops for the condition on ‘director’ and ‘movie_direction’ tables and on ‘movie_direction’ and ‘movie_genre’ tables.

For Hash Join:

Query:

Show work_mem



```
show work_mem
```

	work_mem text
1	4MB

The maximum amount of memory that will be allocated to any query is 4MB
Size of each page is = 8KB

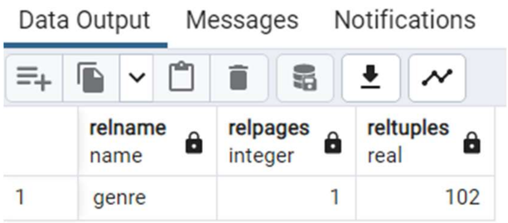
Number of Buffer Pages (BP) = 4MB/8KB = 512 buffer pages

<u>Query:</u>

select relname,relpages,reltuples

from pg_class

where relname='genre';



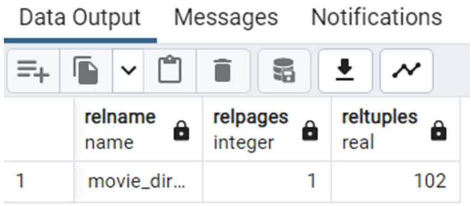
	relname name	relpages integer	reltuples real
1	genre	1	102

Here no of pages in outer relation (M) = 1

select relname,relpages,reltuples

from pg_class

where relname='movie_direction';



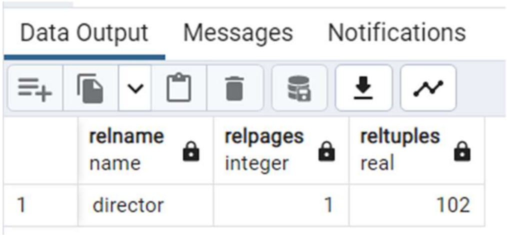
	relname name	relpages integer	reltuples real
1	movie_dir...	1	102

Here no of pages in outer relation (M) = 1

select relname,relpages,reltuples

from pg_class

where relname=' director';



	relname name	relpages integer	reltuples real
1	director	1	102

Here no of pages in outer relation (M) = 1

The resultant query's outer relation from the Join can always fit totally in the buffer memory. Therefore, in this case, the database will use a Hash Join. In this instance, a hash table will be built over the outer relation director, where each hash value will have its corresponding tuples owing to the join property's hash function. Later, we'll use the same hashing technique to hash the join property of the inner relation and search for matches in the outer relation's hash table using the same hashing mechanism.

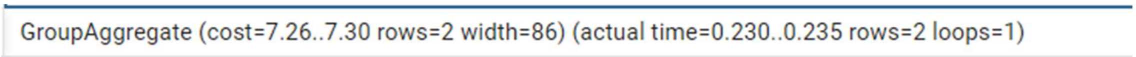
For Nested Loop:

The director and movie_direction are the outer and inner relations, respectively, of the join in this first nested loop. Since the attribute 'dir_id' is a primary key in pgAdmin, an index will already have been created over it. This will allow the database to conduct nested loop operations. The tables movie_direction and movie_genre are the outside and inner relations of the join in the second nested loop, respectively. Since the attribute 'mov_id' is a primary key in pgAdmin, an index will already be built over it, allowing the database to conduct nested loop operations. Since the index was already present by default and just a single search will be required to find the 'dir_id' and 'mov_id' of the tuples of the inner relations, respectively, pgAdmin employs a nested loop in this case.

Because the index is already built over the required attributes (dir id and (mov id), it does not employ alternative joins like hash join or sort merge for these specific cases. For a merge join, there is no requirement to sort the outer relation tuples by attributes or to create a hash table for a hash join.

The estimated cost to run the Query: 7.30

Actual time to run the query: 0.235



```
GroupAggregate (cost=7.26..7.30 rows=2 width=86) (actual time=0.230..0.235 rows=2 loops=1)
```

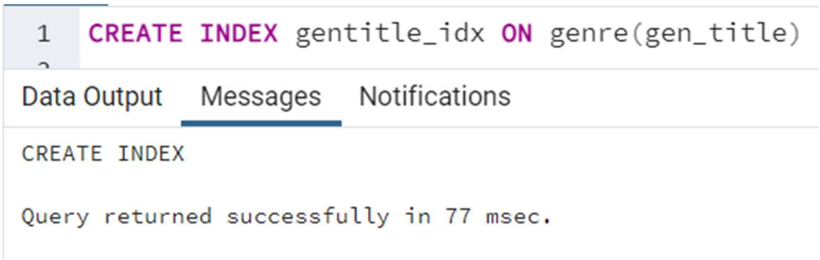
Improve the Performance of the Query:

In order to improve the query performance, we can create a clustered index. Here in this query, we can create index over the attribute ‘gen_title’ on ‘genre’ table to enhance the query performance in this case. Indexes pointing to the data will place them in a more logical order. As a result, the database may match the predicate condition gen title = "Action" by using the index scan rather than the sequential scan. The creation of indexes for additional attributes won't increase the query's efficiency because they won't hasten the data-scanning procedure required to filter the attribute condition. In that case, the index won't match the predicate.

To create the Index:

Query:

CREATE INDEX gentitle_idx ON genre(gen_title)



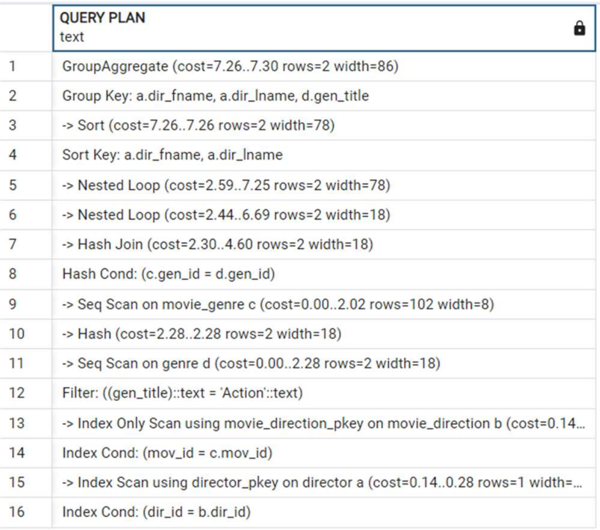
```
CREATE INDEX gentitle_idx ON genre(gen_title)
```

```
CREATE INDEX

Query returned successfully in 77 msec.
```

After Creating Index:

Query:

```
SELECT a.dir_fname,a.dir_lname, d.gen_title,count(d.gen_title)
FROM director a
JOIN movie_direction b
ON a.dir_id = b.dir_id
JOIN movie_genre c
ON b.mov_id = c.mov_id
JOIN genre d
ON c.gen_id = d.gen_id
where gen_title = 'Action'
GROUP BY dir_fname, dir_lname,gen_title
ORDER BY dir_fname,dir_lname;
```



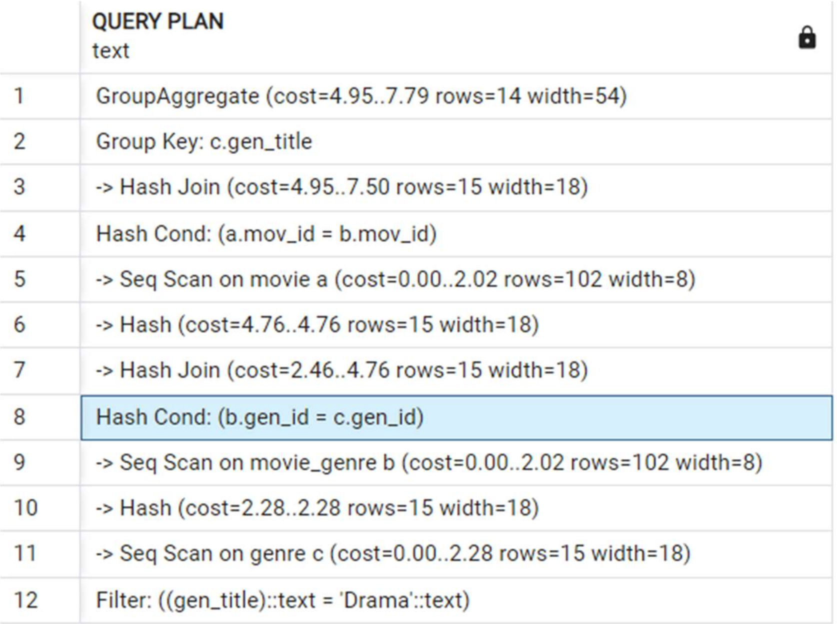
	QUERY PLAN text
1	GroupAggregate (cost=7.26..7.30 rows=2 width=86)
2	Group Key: a.dir_fname, a.dir_lname, d.gen_title
3	-> Sort (cost=7.26..7.26 rows=2 width=78)
4	Sort Key: a.dir_fname, a.dir_lname
5	-> Nested Loop (cost=2.59..7.25 rows=2 width=78)
6	-> Nested Loop (cost=2.44..6.69 rows=2 width=18)
7	-> Hash Join (cost=2.30..4.60 rows=2 width=18)
8	Hash Cond: (c.gen_id = d.gen_id)
9	-> Seq Scan on movie_genre c (cost=0.00..2.02 rows=102 width=8)
10	-> Hash (cost=2.28..2.28 rows=2 width=18)
11	-> Seq Scan on genre d (cost=0.00..2.28 rows=2 width=18)
12	Filter: ((gen_title)::text = 'Action'::text)
13	-> Index Only Scan using movie_direction_pkey on movie_direction b (cost=0.14...
14	Index Cond: (mov_id = c.mov_id)
15	-> Index Scan using director_pkey on director a (cost=0.14..0.28 rows=1 width=...
16	Index Cond: (dir_id = b.dir_id)

Result:

The observation is even after creating index on attribute ‘gen_title’ of ‘genre’ table, there is no change in query performance. As per my understanding if there were more amount of data, postgres might use indexing in order to improve the performance by finding the required data in an efficient way.

Query:(Q9 in section 1)

```
SELECT c.gen_title, AVG(a.mov_time), COUNT(c.gen_title)
 FROM movie a
 JOIN movie_genre b
 ON a.mov_id = b.mov_id
 JOIN  genre c
 ON b.gen_id = c.gen_id
 where c.gen_title = 'Drama'
 GROUP BY gen_title;
```



	QUERY PLAN text
1	GroupAggregate (cost=4.95..7.79 rows=14 width=54)
2	Group Key: c.gen_title
3	-> Hash Join (cost=4.95..7.50 rows=15 width=18)
4	Hash Cond: (a.mov_id = b.mov_id)
5	-> Seq Scan on movie a (cost=0.00..2.02 rows=102 width=8)
6	-> Hash (cost=4.76..4.76 rows=15 width=18)
7	-> Hash Join (cost=2.46..4.76 rows=15 width=18)
8	Hash Cond: (b.gen_id = c.gen_id)
9	-> Seq Scan on movie_genre b (cost=0.00..2.02 rows=102 width=8)
10	-> Hash (cost=2.28..2.28 rows=15 width=18)
11	-> Seq Scan on genre c (cost=0.00..2.28 rows=15 width=18)
12	Filter: ((gen_title)::text = 'Drama'::text)

Join Algorithm Used: Hash Join

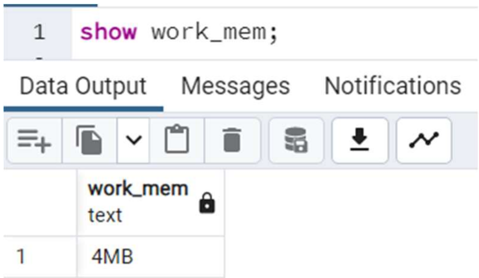
Reason:

Postgres uses Hash join for query that I’ve chosen.

It uses Hash join on ‘movie’and ‘movie_genre’ tables.

For Hash Join:

Show work_mem



```
show work_mem;
```

	work_mem text
1	4MB

The maximum amount of memory that will be allocated to any query is 4MB Size of each page is = 8KB

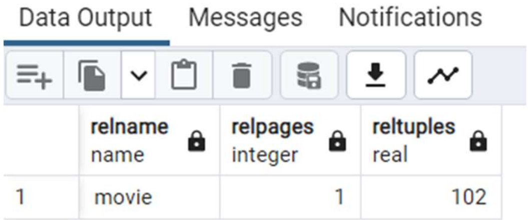
Number of Buffer Pages (BP) = 4MB/8KB = 512 buffer pages

<u>Query:</u>

select relname,relpages,reltuples

from pg_class

where relname='movie';



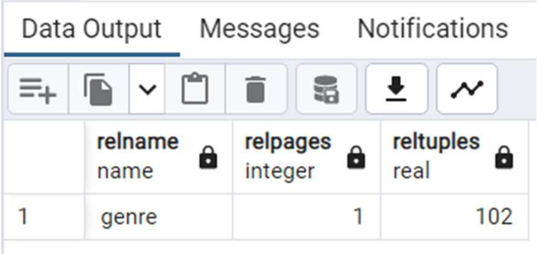
	relname name	relpages integer	reltuples real
1	movie	1	102

Here no of pages in outer relation (M) = 1

select relname,relpages,reltuples

from pg_class

where relname='genre';



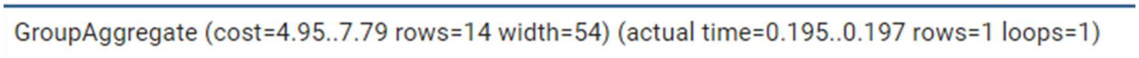
	relname name	relpages integer	reltuples real
1	genre	1	102

Here no of pages in outer relation (M) = 1

In the aforementioned scenarios, the resultant query's outer relation from the Join can completely fit in the buffer memory. The database will therefore attempt to use the Hash Join in this situation. In this instance, a hash table will be built over the outer relation movie, where each hash value will have its own associated hash value depending on the join property's (mov id) hash function. Later, using the same hashing technique, we will hash the join attribute of the inner relation (movie genre) and search for matches in the outer relation's hash table.

The estimated cost to run the Query: 7.79

Actual time to run the query: 0.195



```
GroupAggregate (cost=4.95..7.79 rows=14 width=54) (actual time=0.195..0.197 rows=1 loops=1)
```

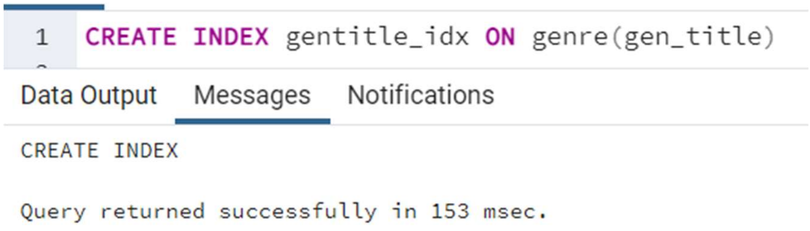
Improve the Performance of the Query:

A clustered index can be built in order to enhance query performance. To improve the query performance in this situation, we can establish an index over the genre table's attribute "gen_title" in this query. With indexes referring to them, the data will be organized in a more sensible order. As a result, rather than performing the sequential scan, the database may match the predicate condition gen title = "Drama" by using the index scan. Indexes for other attributes won't improve query performance because they won't speed up the process of reading the data to filter the attribute condition. The index won't match the predicate in such case.

TO create the Index:

Query:

CREATE INDEX gentitle_idx ON genre(gen_title)



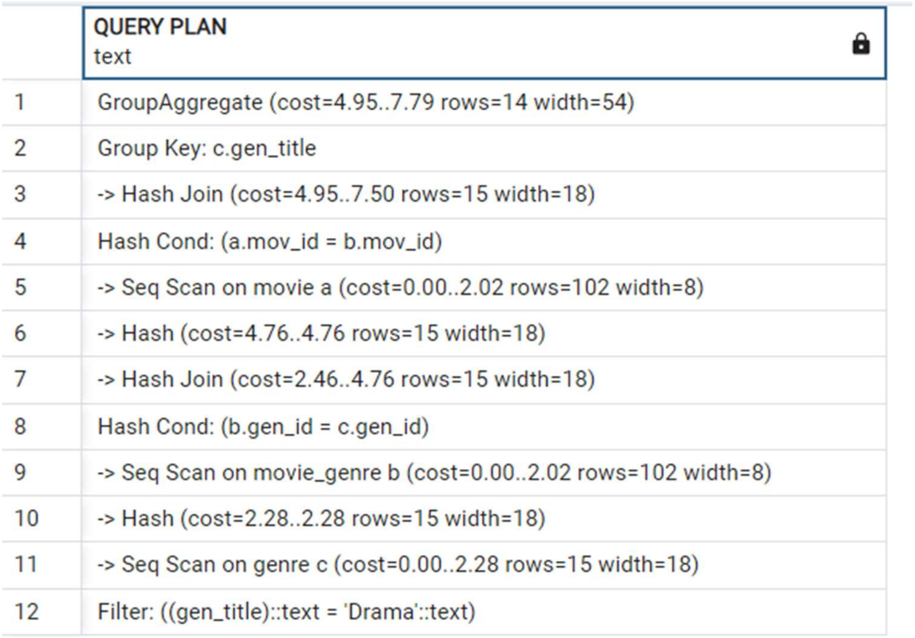
```
CREATE INDEX gentitle_idx ON genre(gen_title)
```

```
CREATE INDEX

Query returned successfully in 153 msec.
```

After Creating Index:

Query:

```
SELECT c.gen_title, AVG(a.mov_time), COUNT(c.gen_title)
 FROM movie a
 JOIN movie_genre b
 ON a.mov_id = b.mov_id
 JOIN  genre c
 ON b.gen_id = c.gen_id
  where c.gen_title = 'Drama'
  GROUP BY gen_title;
```



	QUERY PLAN text
1	GroupAggregate (cost=4.95..7.79 rows=14 width=54)
2	Group Key: c.gen_title
3	-> Hash Join (cost=4.95..7.50 rows=15 width=18)
4	Hash Cond: (a.mov_id = b.mov_id)
5	-> Seq Scan on movie a (cost=0.00..2.02 rows=102 width=8)
6	-> Hash (cost=4.76..4.76 rows=15 width=18)
7	-> Hash Join (cost=2.46..4.76 rows=15 width=18)
8	Hash Cond: (b.gen_id = c.gen_id)
9	-> Seq Scan on movie_genre b (cost=0.00..2.02 rows=102 width=8)
10	-> Hash (cost=2.28..2.28 rows=15 width=18)
11	-> Seq Scan on genre c (cost=0.00..2.28 rows=15 width=18)
12	Filter: ((gen_title)::text = 'Drama'::text)

Result:

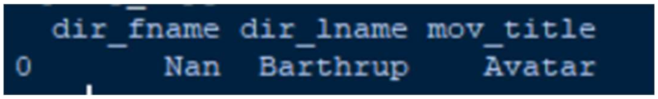
The observation is even after creating index on ‘gen_title’ attribute of ‘genre’ table, there is no change in query performance. As per my understanding if there were more amount of data, postgres might use indexing in order to improve the performance by finding the required data in an efficient way.

3. Query Plan:

Query :(Q2 in Section1)

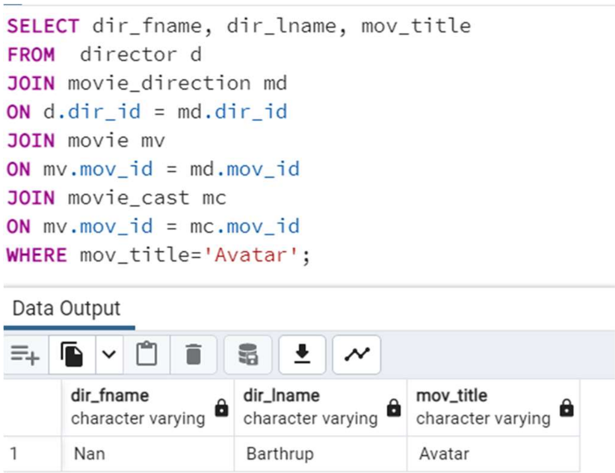
```
SELECT dir_fname, dir_lname, mov_title
FROM  director d
JOIN movie_direction md
ON d.dir_id = md.dir_id
JOIN movie mv
ON mv.mov_id = md.mov_id
JOIN movie_cast mc
ON mv.mov_id = mc.mov_id
WHERE mov_title='Avatar';
```

Output using Python:



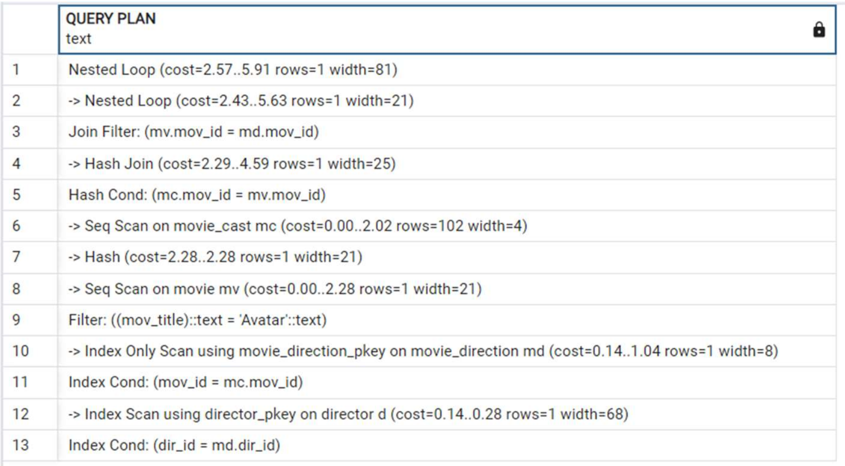
```
  dir_fname dir_lname mov_title
0       Nan  Barthrup    Avatar
```

Output using pgAdmin:



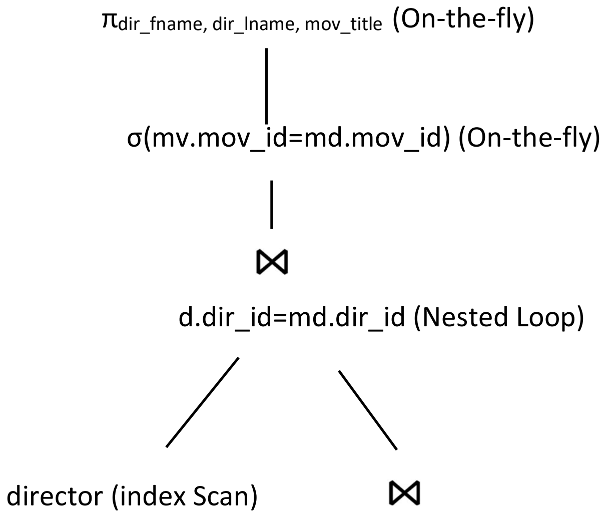
```
SELECT dir_fname, dir_lname, mov_title
FROM  director d
JOIN movie_direction md
ON d.dir_id = md.dir_id
JOIN movie mv
ON mv.mov_id = md.mov_id
JOIN movie_cast mc
ON mv.mov_id = mc.mov_id
WHERE mov_title='Avatar';
```

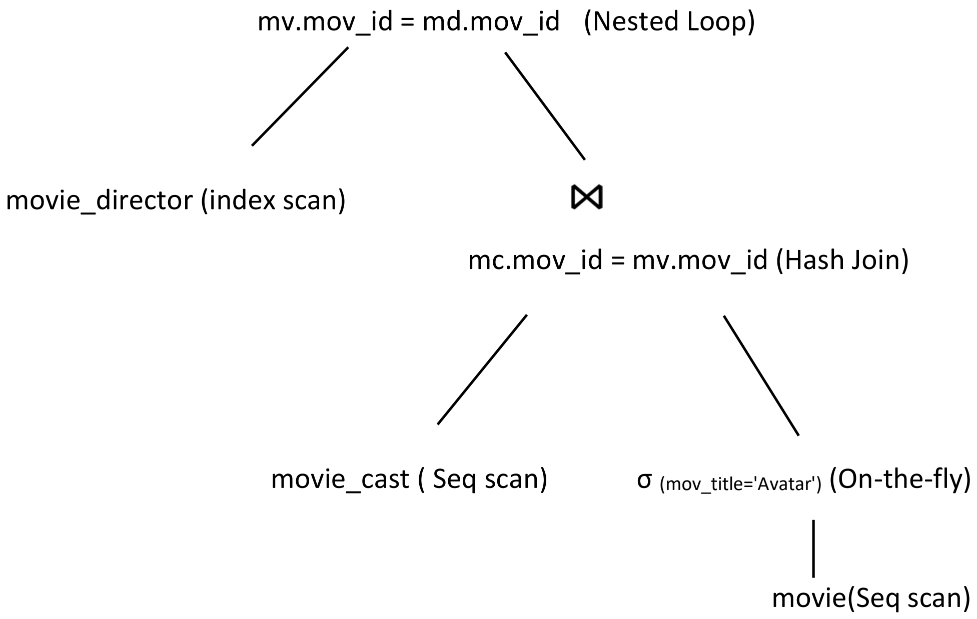
	dir_fname character varying	dir_lname character varying	mov_title character varying
1	Nan	Barthrup	Avatar



	QUERY PLAN text
1	Nested Loop (cost=2.57..5.91 rows=1 width=81)
2	-> Nested Loop (cost=2.43..5.63 rows=1 width=21)
3	Join Filter: (mv.mov_id = md.mov_id)
4	-> Hash Join (cost=2.29..4.59 rows=1 width=25)
5	Hash Cond: (mc.mov_id = mv.mov_id)
6	-> Seq Scan on movie_cast mc (cost=0.00..2.02 rows=102 width=4)
7	-> Hash (cost=2.28..2.28 rows=1 width=21)
8	-> Seq Scan on movie mv (cost=0.00..2.28 rows=1 width=21)
9	Filter: ((mov_title)::text = 'Avatar'::text)
10	-> Index Only Scan using movie_direction_pkey on movie_direction md (cost=0.14..1.04 rows=1 width=8)
11	Index Cond: (mov_id = mc.mov_id)
12	-> Index Scan using director_pkey on director d (cost=0.14..0.28 rows=1 width=68)
13	Index Cond: (dir_id = md.dir_id)

Physical Query Plan:


πdir_fname, dir_lname, mov_title (On-the-fly)
σ(mv.mov_id=md.mov_id) (On-the-fly)
⋈
d.dir_id=md.dir_id (Nested Loop)
director (index Scan)
⋈


mv.mov_id = md.mov_id (Nested Loop)
movie_director (index scan)
⋈
mc.mov_id = mv.mov_id (Hash Join)
movie_cast (Seq scan)
σ (mov_title='Avatar') (On-the-fly)
movie(Seq scan)

Join Algorithm used by Postgres:

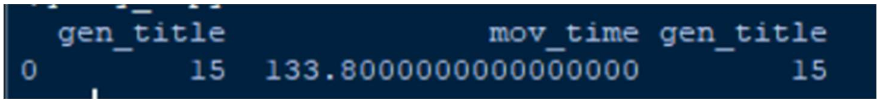
For this query, Postgres uses Hash join for the condition on movie_cast and movie tables where it uses sequential scan on both of the tables movie_cast and movie. Postgres uses Hash join because the resultant query's outer relation director from the join can completely fit in the main buffer memory (which is less than 512 pages) and it’s also an equi-join between both the relations.

It also uses two nested loops where it uses Index Only scan for movie_direction table and Index scan on director table. The index scan helps to search according to the index created by default as pgAdmin constructs index over the respective attributes dir_id, mov_id as they are primary keys. Later since we have a fliter condition [WHERE mov_title='Avatar'], the data/content for scanning will be reduced. So, rather than sequential scan in Hash condition, Postgres will try to implement index scanning and do Nested loop to join the tables.

Query(Q9 in Section 1):

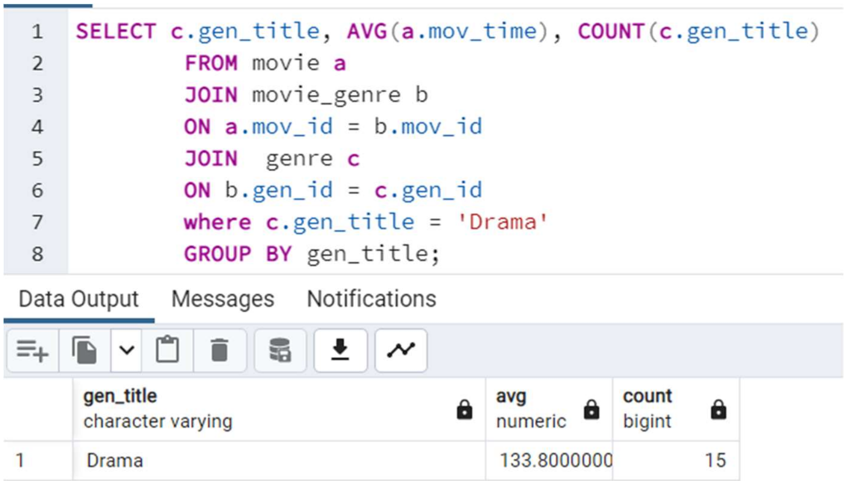
```
SELECT c.gen_title, AVG(a.mov_time), COUNT(c.gen_title)
FROM movie a
JOIN movie_genre b
ON a.mov_id = b.mov_id
JOIN  genre c
ON b.gen_id = c.gen_id
where c.gen_title = 'Drama'
GROUP BY gen_title;
```

Output using Python:



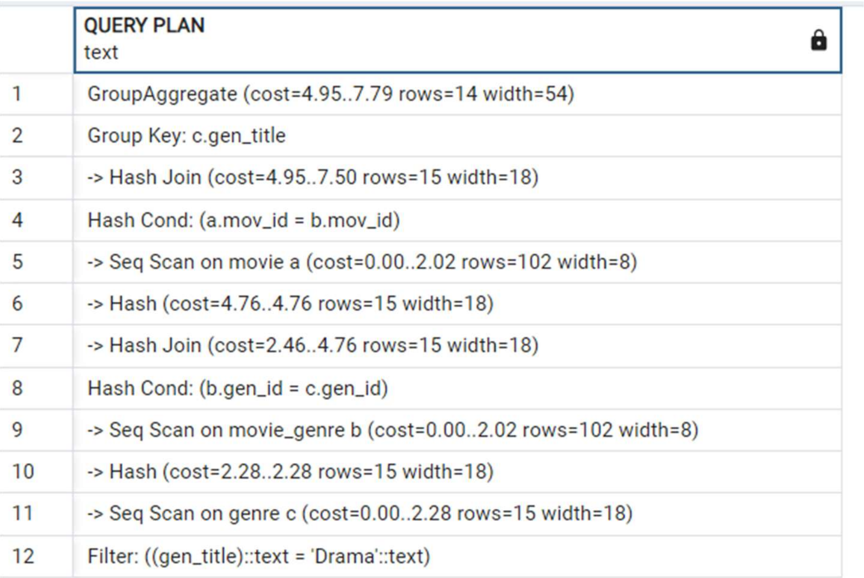
```
  gen_title                 mov_time gen_title
0        15  133.8000000000000000        15
```

Output using pgAdmin:



```sql
SELECT c.gen_title, AVG(a.mov_time), COUNT(c.gen_title)
        FROM movie a
        JOIN movie_genre b
        ON a.mov_id = b.mov_id
        JOIN  genre c
        ON b.gen_id = c.gen_id
        where c.gen_title = 'Drama'
        GROUP BY gen_title;
```

	gen_title character varying	avg numeric	count bigint
1	Drama	133.8000000	15

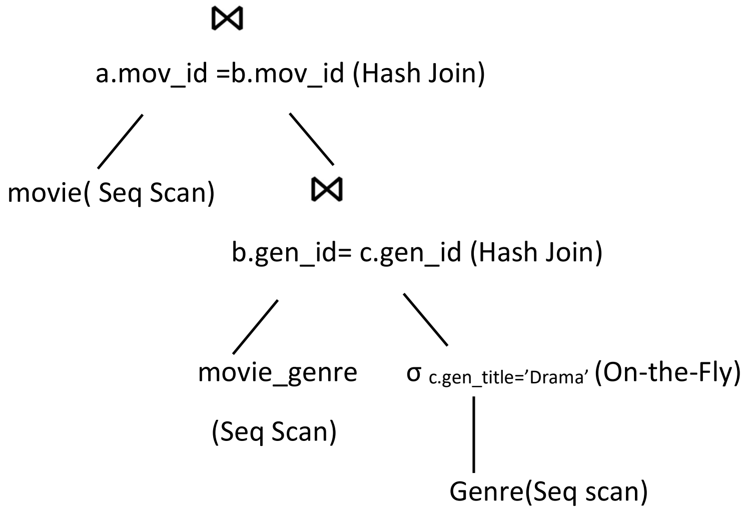


	QUERY PLAN text
1	GroupAggregate (cost=4.95..7.79 rows=14 width=54)
2	Group Key: c.gen_title
3	-> Hash Join (cost=4.95..7.50 rows=15 width=18)
4	Hash Cond: (a.mov_id = b.mov_id)
5	-> Seq Scan on movie a (cost=0.00..2.02 rows=102 width=8)
6	-> Hash (cost=4.76..4.76 rows=15 width=18)
7	-> Hash Join (cost=2.46..4.76 rows=15 width=18)
8	Hash Cond: (b.gen_id = c.gen_id)
9	-> Seq Scan on movie_genre b (cost=0.00..2.02 rows=102 width=8)
10	-> Hash (cost=2.28..2.28 rows=15 width=18)
11	-> Seq Scan on genre c (cost=0.00..2.28 rows=15 width=18)
12	Filter: ((gen_title)::text = 'Drama'::text)

Physical Query Plan:

$\Upsilon_{\text{gen\_title, avg(a.mov\_time),count(c.gen\_title)}}$ (Group Aggregate)

$\pi_{\text{gen\_title, avg(a.mov\_time),count(c.gen\_title)}}$ (On-the-Fly)


⋈
a.mov_id =b.mov_id (Hash Join)
movie(Seq Scan)
⋈
b.gen_id= c.gen_id (Hash Join)
movie_genre
(Seq Scan)
σ c.gen_title='Drama' (On-the-Fly)
Genre(Seq scan)

Join Algorithm used by Postgres:

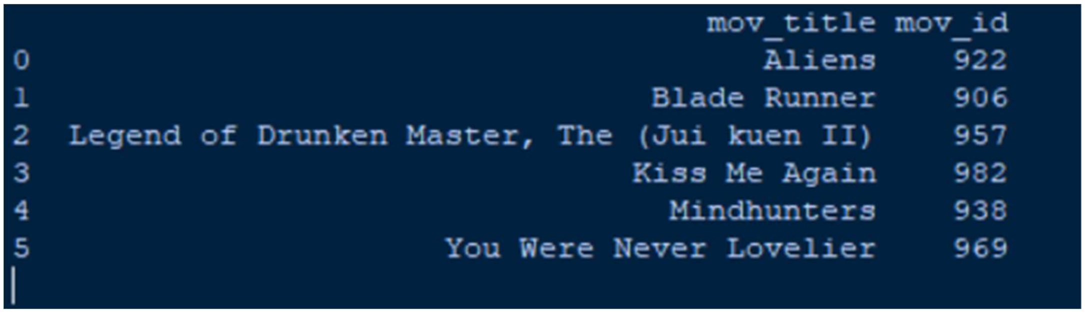
For this query, Postgres uses Hash join for the condition on movie and movie_genre tables where it uses sequential scan on both of the relations movie_genre and movie. Postgres uses Hash join because the resultant query's outer relation director from the join can completely fit in the main buffer memory (which is less than 512 pages) and it's also an equi-join between both the relations.

4. Visualization:

1. **write a SQL query to search for movies that do not have any ratings. Return movie title. (Query 4 in section 1).**

ANS:

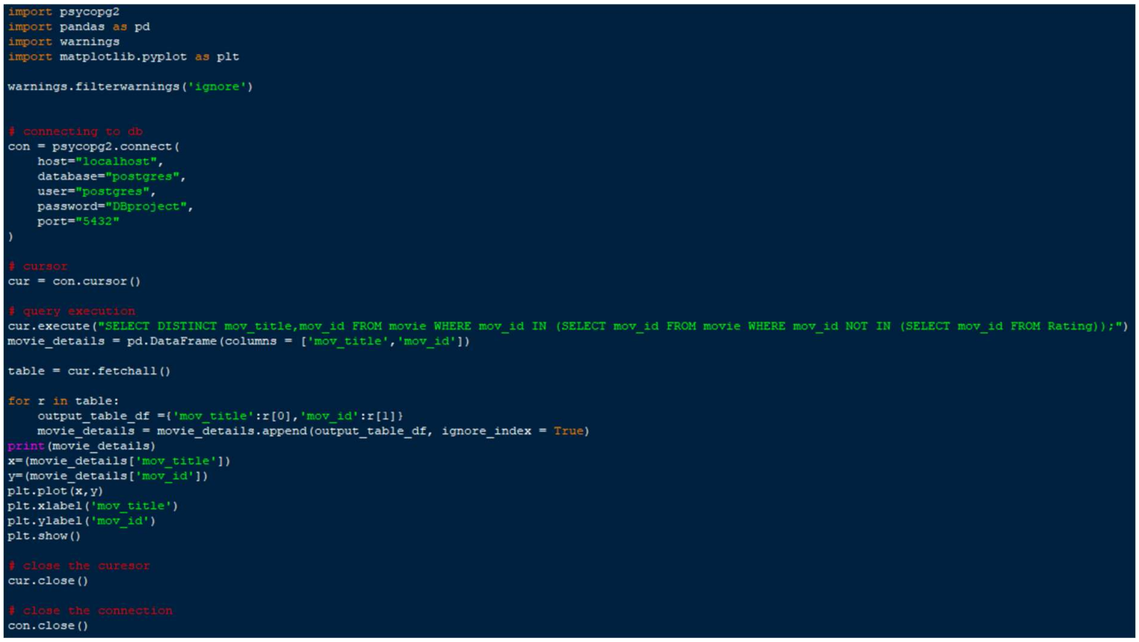
```
SELECT DISTINCT mov_title, mov_id
FROM movie
WHERE mov_id IN (
SELECT mov_id
FROM movie
WHERE mov_id NOT IN (
SELECT mov_id FROM Rating));
```



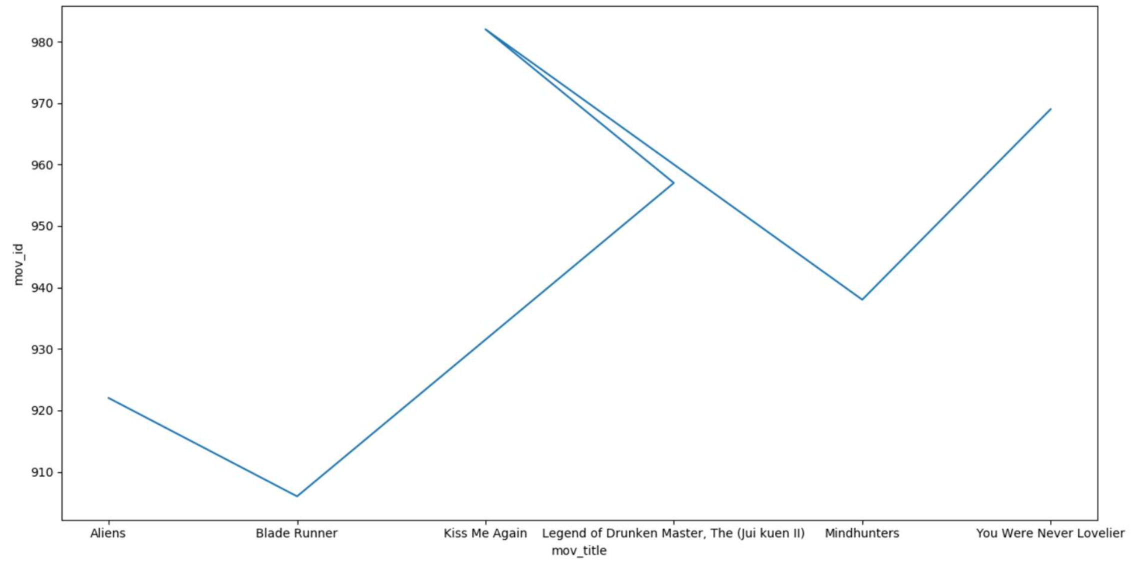
```
                                          mov_title mov_id
0                                            Aliens    922
1                                      Blade Runner    906
2  Legend of Drunken Master, The (Jui kuen II)    957
3                                     Kiss Me Again    982
4                                       Mindhunters    938
5                            You Were Never Lovelier    969
```

Explanation:

The query that I have chosen here gives the output of the movie titles that do not have any ratings along with the movie id. So, I wanted to implement the visualization of data for this query. With the help of matplotlib in python, I have developed the graph for data visualization which has mov_title attribute on x-axis and mov_id on the y-axis. So, we can see information of all the movie titles and their respective movie id’s pointing on the graph. Below is the screenshot of the code.



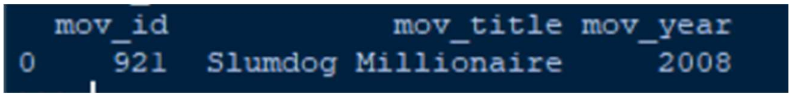
```python
import psycopg2
import pandas as pd
import warnings
import matplotlib.pyplot as plt

warnings.filterwarnings('ignore')


# connecting to db
con = psycopg2.connect(
    host="localhost",
    database="postgres",
    user="postgres",
    password="DBproject",
    port="5432"
)

# cursor
cur = con.cursor()

# query execution
cur.execute("SELECT DISTINCT mov_title,mov_id FROM movie WHERE mov_id IN (SELECT mov_id FROM movie WHERE mov_id NOT IN (SELECT mov_id FROM Rating));")
movie_details = pd.DataFrame(columns = ['mov_title','mov_id'])

table = cur.fetchall()

for r in table:
    output_table_df ={'mov_title':r[0],'mov_id':r[1]}
    movie_details = movie_details.append(output_table_df, ignore_index = True)
print(movie_details)
x=(movie_details['mov_title'])
y=(movie_details['mov_id'])
plt.plot(x,y)
plt.xlabel('mov_title')
plt.ylabel('mov_id')
plt.show()

# close the curesor
cur.close()

# close the connection
con.close()
```



2. **write a SQL query to find the movie titles that starts with the word 'Slumdog'. Sort the result order by movie year. Return movie ID, movie title and movie release year. (Q7 in Section 1)**

ANS:

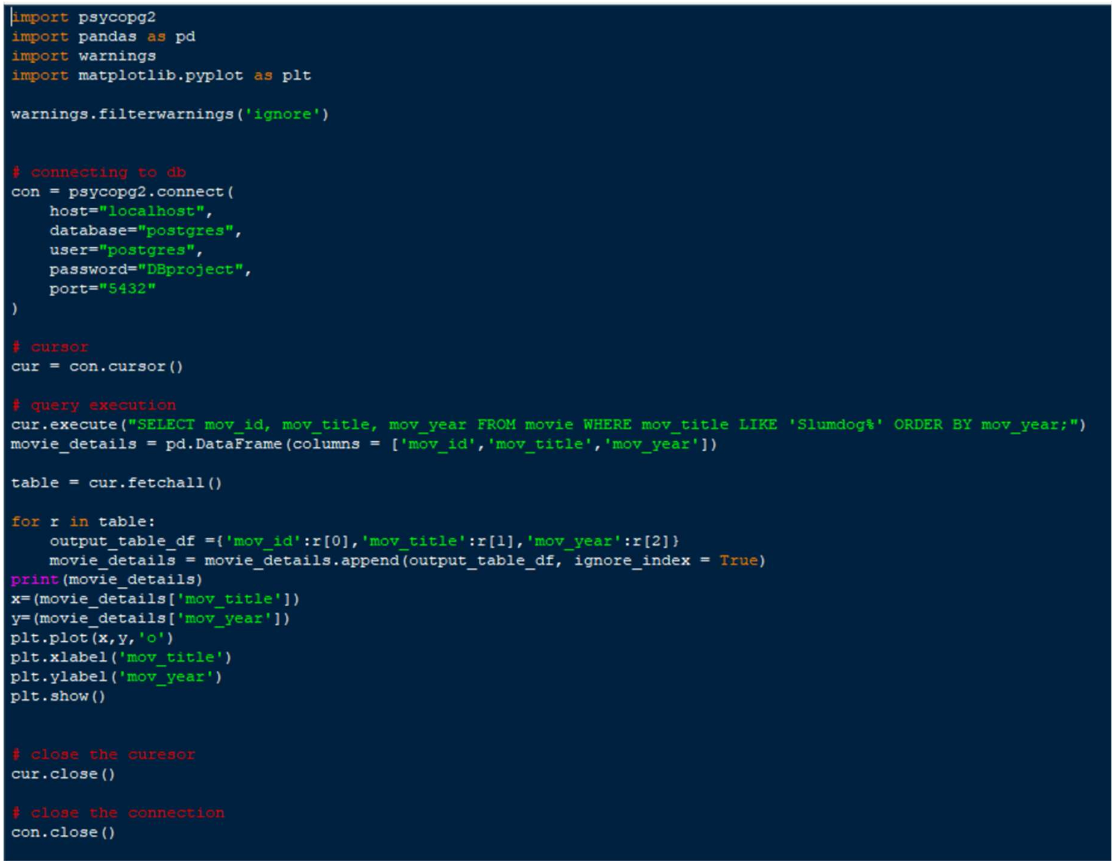
```
SELECT mov_id, mov_title, mov_year
FROM movie
WHERE mov_title LIKE 'Slumdog%'
ORDER BY mov_year;
```



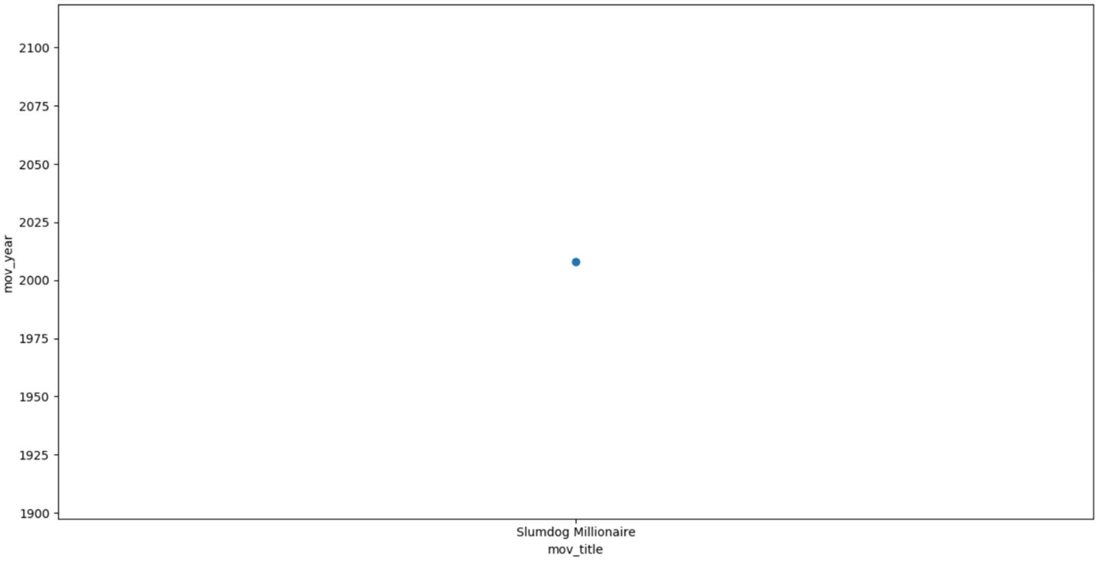
```
  mov_id            mov_title mov_year
0    921  Slumdog Millionaire     2008
```

Explanation:

The query that I have chosen here gives the output of the movie title that has name with Slumdog along with the movie id and movie year. So, I wanted to implement the visualization of data for this query. With the help of matplotlib in python, I have developed the graph for data visualization which has mov_title attribute on x-axis and mov_year on the y-axis. So, we can see information of all the movie titles and their respective release date for the movie pointing on the graph. Since there is only one movie with that name we can see the one dot referring to the output of the query. Below is the screenshot of the code.



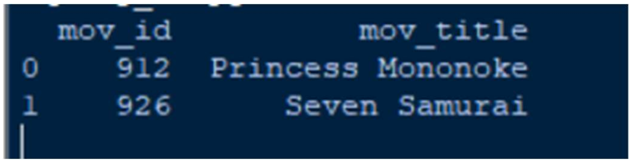
```
import psycopg2
import pandas as pd
import warnings
import matplotlib.pyplot as plt

warnings.filterwarnings('ignore')


# connecting to db
con = psycopg2.connect(
    host="localhost",
    database="postgres",
    user="postgres",
    password="DBproject",
    port="5432"
)

# cursor
cur = con.cursor()

# query execution
cur.execute("SELECT mov_id, mov_title, mov_year FROM movie WHERE mov_title LIKE 'Slumdog%' ORDER BY mov_year;")
movie_details = pd.DataFrame(columns = ['mov_id','mov_title','mov_year'])

table = cur.fetchall()

for r in table:
    output_table_df ={'mov_id':r[0],'mov_title':r[1],'mov_year':r[2]}
    movie_details = movie_details.append(output_table_df, ignore_index = True)
print(movie_details)
x=(movie_details['mov_title'])
y=(movie_details['mov_year'])
plt.plot(x,y,'o')
plt.xlabel('mov_title')
plt.ylabel('mov_year')
plt.show()


# close the curesor
cur.close()

# close the connection
con.close()
```



3. **write a SQL query to find those movies, which were released before 1998 and language is Japanese. (Query 10 in Section 2).**

ANS:

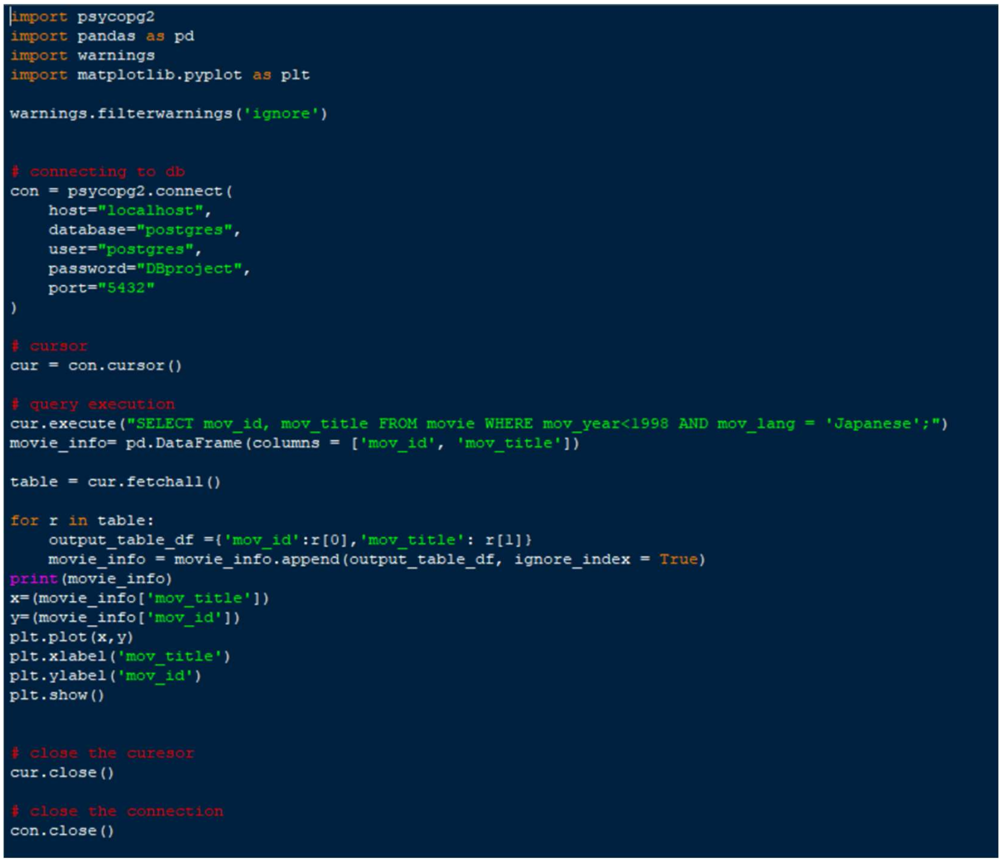
```
SELECT mov_id, mov_title
FROM movie
WHERE mov_year<1998
AND mov_lang = 'Japanese'
```



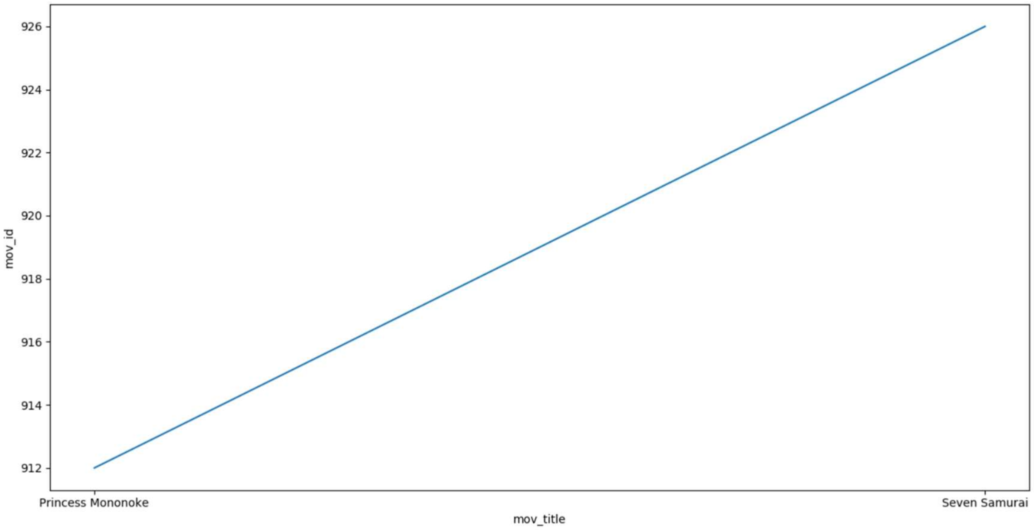
```
   mov_id          mov_title
0     912  Princess Mononoke
1     926      Seven Samurai
```

Explanation:

The query that I have chosen here gives the output of the movie title and movie id whose language of the movie is Japanese. So, I wanted to implement the visualization of data for this query. With the help of matplotlib in python, I have developed the graph for data visualization which has mov_title attribute on x-axis and mov_id on the y-axis. So, we can see information of all the movie titles and their respective movie id's for the movie pointing on the graph. From the output we can see that there are two movies and the line represents that in the graph. Below is the screenshot of the code.



```python
import psycopg2
import pandas as pd
import warnings
import matplotlib.pyplot as plt

warnings.filterwarnings('ignore')


# connecting to db
con = psycopg2.connect(
    host="localhost",
    database="postgres",
    user="postgres",
    password="DBproject",
    port="5432"
)

# cursor
cur = con.cursor()

# query execution
cur.execute("SELECT mov_id, mov_title FROM movie WHERE mov_year<1998 AND mov_lang = 'Japanese';")
movie_info= pd.DataFrame(columns = ['mov_id', 'mov_title'])

table = cur.fetchall()

for r in table:
    output_table_df ={'mov_id':r[0],'mov_title': r[1]}
    movie_info = movie_info.append(output_table_df, ignore_index = True)
print(movie_info)
x=(movie_info['mov_title'])
y=(movie_info['mov_id'])
plt.plot(x,y)
plt.xlabel('mov_title')
plt.ylabel('mov_id')
plt.show()


# close the curesor
cur.close()

# close the connection
con.close()
```



5. Presentation:

Presentation Link:

https://drive.google.com/file/d/18e_3f8ykze-qPfD0KeDDbX9ndWB__YxL/view?usp=sharing