

# BDS-2 Project Assignment – SQL and Security Basics

Ivan Lohunkov **241069**

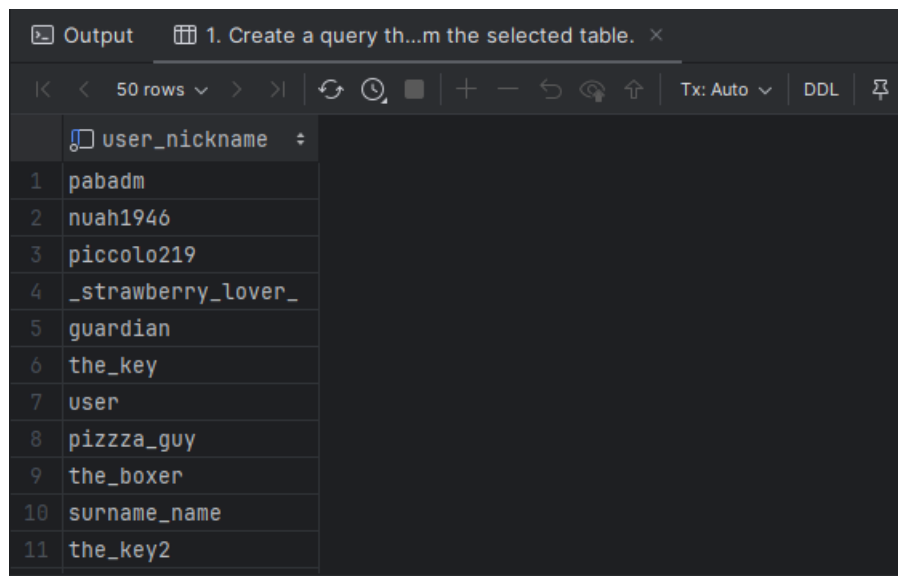
19. listopadu 2023

# 1 Part SQL

## 1.1 Create a query that will retrieve only selected columns from the selected table

In this query, we select `user_nickname` from the `user` table.

```
SELECT user_nickname FROM "user";
```



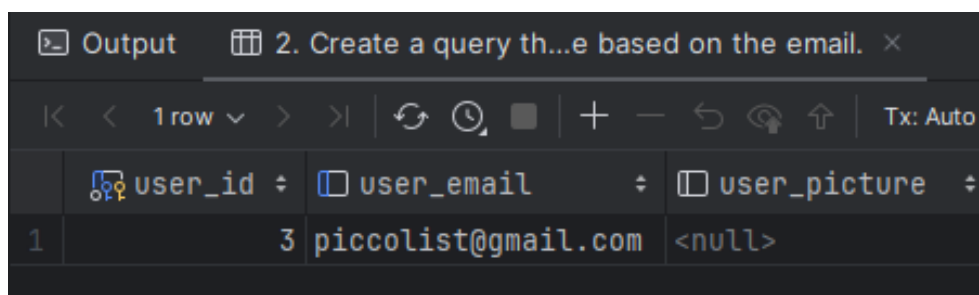
The screenshot shows a database interface with a tab titled "1. Create a query th...m the selected table. x". Below the tab is a toolbar with navigation and execution icons. The main area displays a table with the column `user_nickname`. The table contains 11 rows of data.

	user_nickname
1	pabadm
2	nuah1946
3	piccolo219
4	_strawberry_lover_
5	guardian
6	the_key
7	user
8	pizza_guy
9	the_boxer
10	surname_name
11	the_key2

## 1.2 Create a query that will select a user/person or similar table based on the email

For this task, we had to choose a row from the table, based on the e-mails.

```
SELECT * FROM user_detail  
WHERE user_email = 'piccolist@gmail.com';
```



The screenshot shows a database interface with a tab titled "2. Create a query th...e based on the email. x". Below the tab is a toolbar with navigation and execution icons. The main area displays a table with three columns: `user_id`, `user_email`, and `user_picture`. The table contains 1 row of data.

	user_id	user_email	user_picture
1	3	piccolist@gmail.com	<null>

### 1.3 Create at least one UPDATE, INSERT, DELETE, and ALTER TABLE query

In this assignment we had to make UPDATE, INSERT, DELETE, ALTER TABLE queries. The following will be written under each query separately.

#### 1.3.1 UPDATE

Here a different movie\_budget for 5 different films has been added so that the difference can be seen in further queries.

```
UPDATE movie SET movie_budget = 9500000
WHERE movie_id = 1;

UPDATE movie SET movie_budget = 1000
WHERE movie_id = 2;

UPDATE movie SET movie_budget = 600000
WHERE movie_id = 3;

UPDATE movie SET movie_budget = 50000
WHERE movie_id = 4;

UPDATE movie SET movie_budget = 123456
WHERE movie_id = 5;
```

### 1.3.2 INSERT

One comment was added for a film with movie\_id = 4.

```
INSERT INTO movie_comments
(user_id, movie_id, comment_text, comment_publish)
VALUES (8,4, 'great!', '2020-01-01');
```

### 1.3.3 DELETE

Here the query removes the country with country\_id = 10.

```
DELETE FROM country WHERE country_id = 10;
```

### 1.3.4 ALTER TABLE

The table is renamed using ALTER TABLE query

```
ALTER TABLE user_detail RENAME TO user_details;

ALTER TABLE user_details RENAME TO user_detail;
```

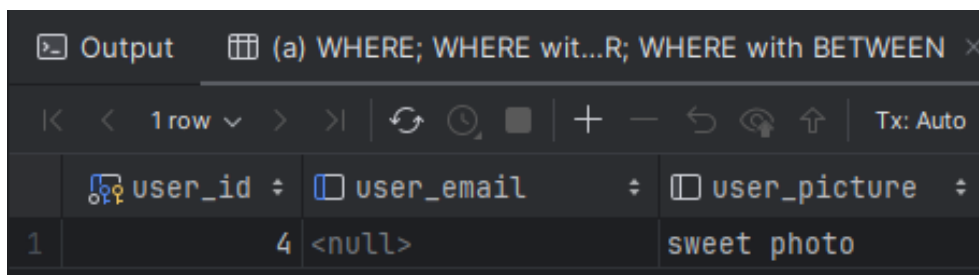
```
[2023-11-18 16:43:52] 1 row retrieved starting from 1 in 65 ms (execution: 23 ms, fetching: 42 ms)
postgres.public> UPDATE movie SET movie_budget = 9500000 WHERE movie_id = 1
[2023-11-18 16:45:05] 1 row affected in 24 ms
postgres.public> UPDATE movie SET movie_budget = 1000 WHERE movie_id = 2
[2023-11-18 16:45:35] 1 row affected in 13 ms
postgres.public> UPDATE movie SET movie_budget = 600000 WHERE movie_id = 3
[2023-11-18 16:46:17] 1 row affected in 14 ms
postgres.public> UPDATE movie SET movie_budget = 50000 WHERE movie_id = 4
[2023-11-18 16:46:19] 1 row affected in 13 ms
postgres.public> UPDATE movie SET movie_budget = 123456 WHERE movie_id = 5
[2023-11-18 16:46:22] 1 row affected in 14 ms
postgres.public> INSERT INTO movie_comments( user_id, movie_id, comment_text, comment_publish)
VALUES (8,4, 'great!', '2020-01-01')
[2023-11-18 16:46:38] 1 row affected in 25 ms
postgres.public> DELETE FROM country WHERE country_id = 10
[2023-11-18 16:46:42] completed in 19 ms
postgres.public> ALTER TABLE user_detail RENAME TO user_details
[2023-11-18 16:46:53] completed in 15 ms
postgres.public> ALTER TABLE user_details RENAME TO user_detail
[2023-11-18 16:46:55] completed in 4 ms
```

## 1.4 Create a series of queries that will separately use the following:

### 1.4.1 WHERE; WHERE with AND; WHERE with OR; WHERE with BETWEEN

In this query, we select all „ \* “ from the `user_detail` table where the `user_picture` column is equal to a specific picture named `sweet photo`.

```
SELECT * FROM user_detail
WHERE user_picture = 'sweet photo';
```

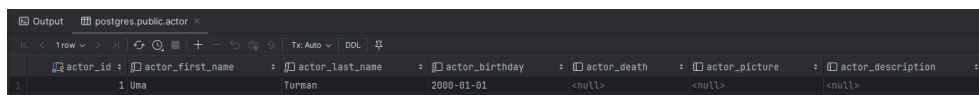


The screenshot shows a database interface with a query window titled "(a) WHERE; WHERE wit...R; WHERE with BETWEEN". The output table has three columns: `user_id`, `user_email`, and `user_picture`. The first row contains the values 4, <null>, and sweet photo.

	user_id	user_email	user_picture
1	4	<null>	sweet photo

Here we search the database in the `actor` table, where we are found by the first and last name of a particular actor.

```
SELECT * FROM actor
WHERE actor_first_name = 'Uma'
AND actor_last_name = 'Turman';
```



The screenshot shows a database interface with a query window titled "postgres.public.actor". The output table has seven columns: `actor_id`, `actor_first_name`, `actor_last_name`, `actor_birthday`, `actor_death`, `actor_picture`, and `actor_description`. The first row contains the values 1, Uma, Turman, 2000-01-01, <null>, <null>, and <null>.

	actor_id	actor_first_name	actor_last_name	actor_birthday	actor_death	actor_picture	actor_description
1	1	Uma	Turman	2000-01-01	<null>	<null>	<null>

There are two options in this query, one is to search by date of birth or by last name.

```
SELECT * FROM director
WHERE director_birthday = '1963-02-27'
OR director_last_name = 'Tarantino';
```

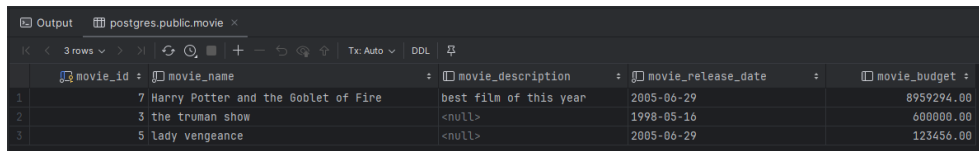


The screenshot shows a database interface with a query window titled "postgres.public.director". The output table has seven columns: `director_id`, `director_first_name`, `director_last_name`, `director_birthday`, `director_death`, `director_picture`, and `director_description`. The first row contains the values 1, Quentin, Tarantino, 1963-02-27, <null>, <null>, and <null>.

	director_id	director_first_name	director_last_name	director_birthday	director_death	director_picture	director_description
1	1	Quentin	Tarantino	1963-02-27	<null>	<null>	<null>

Here we search the `movie_release_date` column from the `movie` table between the release year of the film.

```
SELECT * FROM movie
WHERE movie_release_date
BETWEEN '1998-05-16' AND '2005-06-29';
```



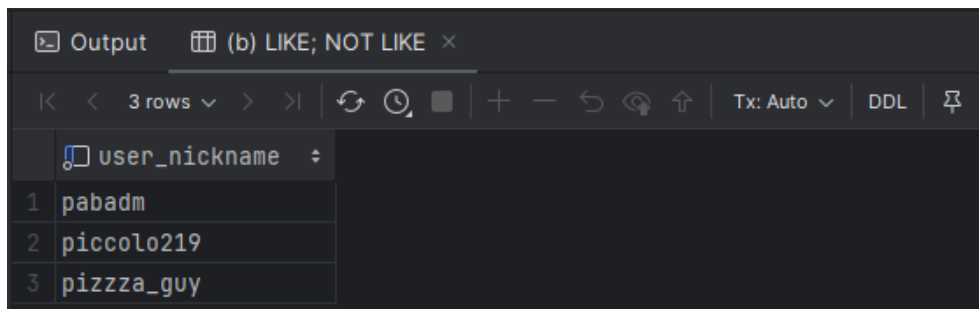
The screenshot shows a database query output window with the title 'postgres.public.movie'. It displays three rows of data. The columns are: movie\_id, movie\_name, movie\_description, movie\_release\_date, and movie\_budget.

	movie_id	movie_name	movie_description	movie_release_date	movie_budget
1	7	Harry Potter and the Goblet of Fire	best film of this year	2005-06-29	8959294.00
2	3	the truman show	<null>	1998-05-16	600000.00
3	5	lady vengeance	<null>	2005-06-29	123456.00

### 1.4.2 LIKE; NOT LIKE

Here, the query is executed so that it looks for `user_nickname` that starts with the letter „p“.

```
SELECT user_nickname FROM "user"
WHERE user_nickname LIKE 'p%';
```



The screenshot shows a database query output window with the title '(b) LIKE; NOT LIKE'. It displays three rows of data. The column is: user\_nickname.

	user_nickname
1	pabadm
2	piccolo219
3	pizza_guy

This is a completely opposite query to the previous one, here we are looking for anything that doesn't start with „the“ in `user_nickname`.

```
SELECT user_nickname FROM "user"
WHERE user_nickname NOT LIKE 'the%';
```

Output postgres.public.user x	
3 rows v   Tx: Auto v   DDL	
	user_nickname
1	pabadm
2	nuah1946
3	piccolo219
4	_strawberry_lover_
5	guardian
6	user
7	pizza_guy
8	surname_name

### 1.4.3 SUBSTRING; TRIM; CONCAT; COALESCE

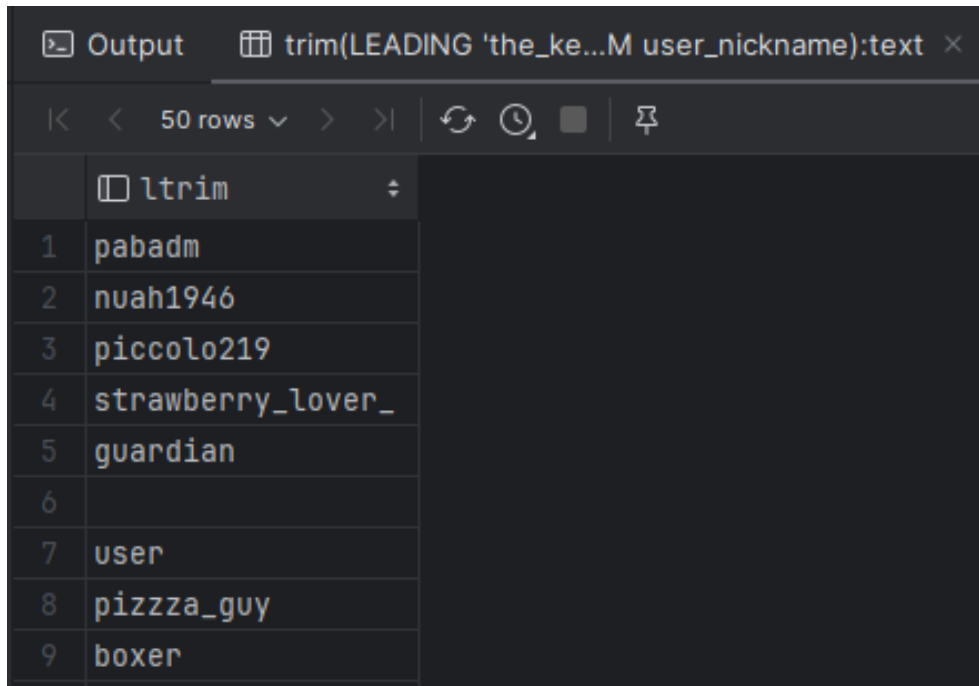
A query with `substring` performs searches that include characters from 1 to 3.

```
SELECT substring(user_nickname, 1, 3)
FROM "user";
```

Output (c) SUBSTRING; TRIM; CONCAT; COALESCE x	
50 rows v	
	substring
1	pab
2	nua
3	pic
4	_st
5	gua
6	the
7	use
8	piz
9	the

`trim` in the query deletes the parts of the text we have selected. In this query, we remove the `LEADING` part of the text at the beginning `the_key`.

```
SELECT trim(LEADING 'the_key' FROM user_nickname)
FROM "user";
```



The screenshot shows a database query output window with a title bar that reads "Output" and "trim(LEADING 'the\_ke...M user\_nickname):text". Below the title bar is a toolbar with navigation icons and a "50 rows" indicator. The main area displays a table with 9 rows. The first column is labeled "ltrim" and the second column is labeled "text". The data in the "text" column is as follows:

	ltrim	text
1	pabadm	
2	nuah1946	
3	piccolo219	
4	strawberry_lover_	
5	guardian	
6		
7	user	
8	pizzza_guy	
9	boxer	

A query with `concat` joins two columns from a table into one.

```
SELECT concat(actor_first_name, ' ', actor_last_name)
AS actor_full_name FROM actor;
```



Output		actor_full_name:text	
		6 rows	
	actor_full_name		
1	Uma Turman		
2	Sacha Baron Cohen		
3	Jim Carrey		
4	Song Kang-ho		
5	Lee Yeong-ae		
6	test actor		

In this case there is `coalesce` in the query, if a particular email has nothing in the picture raises in the „n/a“ column.

```
SELECT coalesce(user_email, 'no email' )
AS Email, coalesce(user_picture, 'n/a' )
AS Picture FROM user_detail;
```



Output <span>budget_min:numeric(10,2) ×</span>	
<div> <div> <div>&lt;</div> <div>&lt;</div> <div>1 row ▾</div> <div>&gt;</div> <div>&gt;</div> </div> <div> <div>↺</div> <div>⌚</div> <div>■</div> <div>📌</div> </div> </div>	
	budget_min ÷
1	1000

```
SELECT max(movie_budget) AS budget_max FROM movie;
```

Output <span>budget_max:numeric(10,2) ×</span>	
<div> <div> <div>&lt;</div> <div>&lt;</div> <div>1 row ▾</div> <div>&gt;</div> <div>&gt;</div> </div> <div> <div>↺</div> <div>⌚</div> <div>■</div> <div>📌</div> </div> </div>	
	budget_max ÷
1	9686575

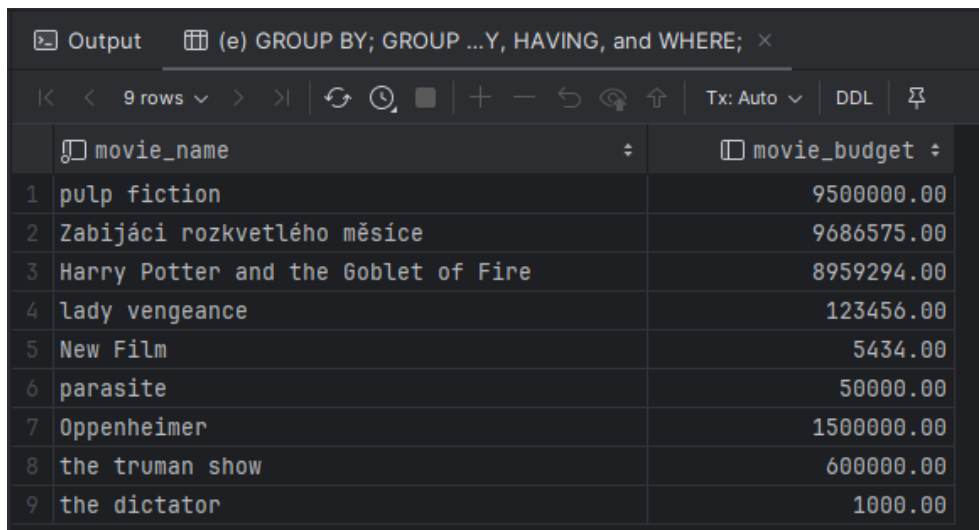
```
SELECT avg(movie_budget) AS budget_avg FROM movie;
```

Output <span>budget_avg:numeric ×</span>	
<div> <div> <div>&lt;</div> <div>&lt;</div> <div>1 row ▾</div> <div>&gt;</div> <div>&gt;</div> </div> <div> <div>↺</div> <div>⌚</div> <div>■</div> <div>📌</div> </div> </div>	
	budget_avg ÷
1	3380639.88888888888889

### 1.4.5 GROUP BY; GROUP BY and HAVING; GROUP BY, HAVING, and WHERE

The GROUP BY clause divides the rows returned from the SELECT statement into groups.

```
SELECT movie_name, movie_budget FROM movie
GROUP BY movie.movie_name, movie.movie_budget;
```



The screenshot shows a database query output window with the title "(e) GROUP BY; GROUP ...Y, HAVING, and WHERE;". The window displays a table with two columns: "movie\_name" and "movie\_budget". The table contains 9 rows of data, sorted by movie\_name. The rows are:

	movie_name	movie_budget
1	pulp fiction	9500000.00
2	Zabijáci rozkvetlého měsíce	9686575.00
3	Harry Potter and the Goblet of Fire	8959294.00
4	lady vengeance	123456.00
5	New Film	5434.00
6	parasite	50000.00
7	Oppenheimer	1500000.00
8	the truman show	600000.00
9	the dictator	1000.00

Here we select those films which have a budget of more than 49000.

```
SELECT movie_budget FROM movie
GROUP BY movie.movie_budget
HAVING movie.movie_budget > 49000;
```

Output postgres.public.movie	
7 rows	
	movie_budget
1	1500000.00
2	9500000.00
3	8959294.00
4	9686575.00
5	50000.00
6	600000.00
7	123456.00

This is similar to the previous one, except here we have added a `WHERE` clause where `movie_release_date = '1994-04-21'`.

```
SELECT movie_budget FROM movie
AS freak WHERE movie_release_date = '1994-04-21'
GROUP BY freak.movie_budget
HAVING freak.movie_budget > 49000;
```

Output postgres.public.movie	
1 row	
movie_budget	
1	9500000.00

#### 1.4.6 UNION ALL / UNION; DISTINCT; COUNT; EXCEPT; INTERSECT

The UNION operator combines result sets of two or more SELECT statements into a single result set.

```
SELECT 'director' AS role,
director_first_name AS first_name,
director_last_name AS last_name FROM director
UNION SELECT 'actor' AS role,
actor_first_name AS first_name,
actor_last_name AS last_name FROM actor;
```

Output (f) UNION ALL / UNIO...NT; EXCEPT; INTERSECT			
11 rows			
	role	first_name	last_name
1	director	Larry	Charles
2	actor	test	actor
3	actor	Song	Kang-ho
4	actor	Sacha	Baron Cohen
5	director	Park	Chan-wook
6	actor	Uma	Turman
7	director	Quentin	Tarantino
8	director	Bong	Joon-ho
9	actor	Lee	Yeong-ae

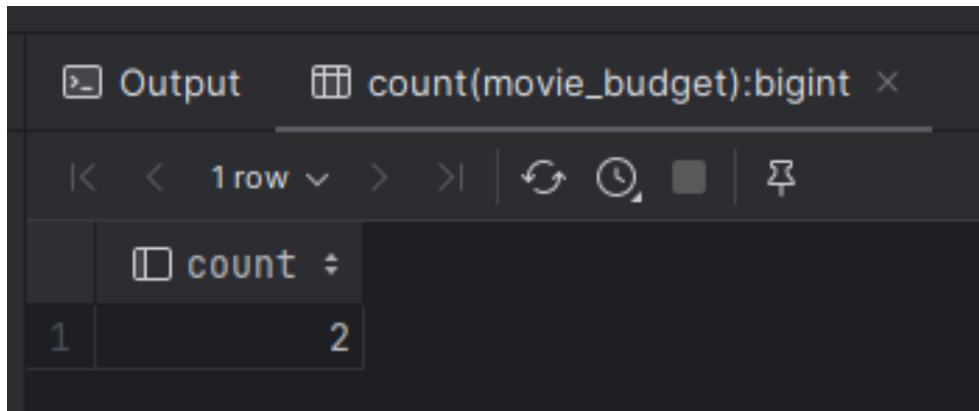
The **DISTINCT** clause is used in the **SELECT** statement to remove duplicate rows from a result set.

```
SELECT DISTINCT user_id
FROM user_has_favorite_actor;
```

Output postgres.public.user_has_favorite_actor	
4 rows	
	user_id
1	3
2	4
3	10
4	7

Here the query counts the number of films that have a budget less than 49000.

```
SELECT count(movie_budget) FROM movie
WHERE movie.movie_budget < 49000;
```

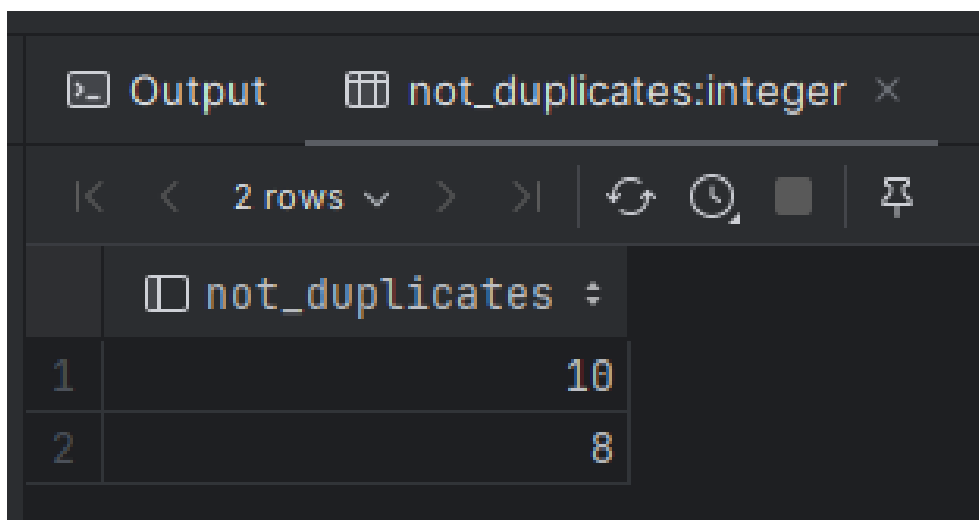


The screenshot shows a database interface with a dark theme. At the top, there's a tab labeled 'Output' and a title bar 'count(movie\_budget):bigint'. Below the title bar is a navigation bar with icons for back, forward, and search, and a dropdown menu showing '1 row'. The main area displays a table with one column named 'count' and one row with the value '2'.

	count
1	2

The EXCEPT operator returns distinct rows from the first (left) query that are not in the output of the second (right) query.

```
SELECT user_id AS not_duplicates
FROM user_has_viewed_movie
EXCEPT SELECT user_id FROM user_has_wished_movie;
```



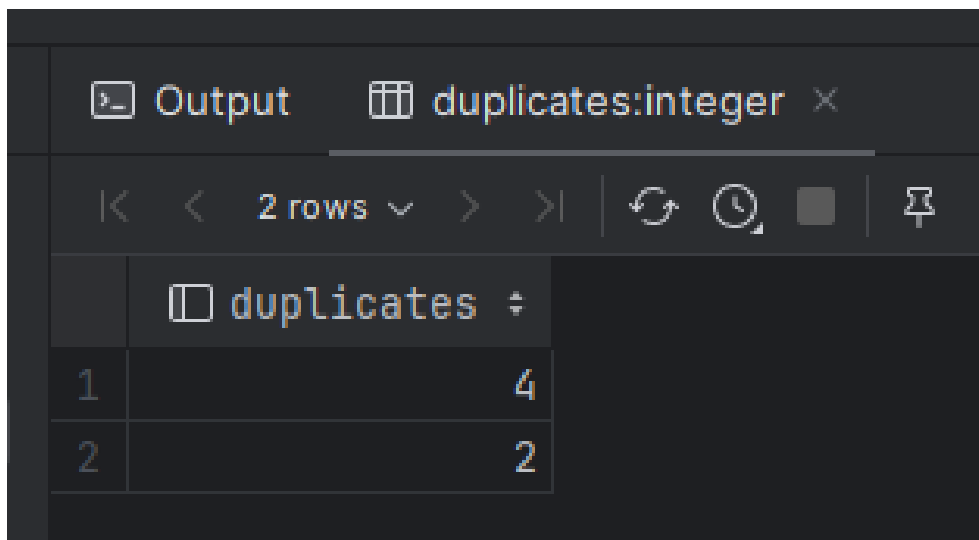
The screenshot shows a database interface with a dark theme. At the top, there's a tab labeled 'Output' and a title bar 'not\_duplicates:integer'. Below the title bar is a navigation bar with icons for back, forward, and search, and a dropdown menu showing '2 rows'. The main area displays a table with one column named 'not\_duplicates' and two rows with values '10' and '8'.

	not_duplicates
1	10
2	8



The INTERSECT operator returns any rows that are available in both result sets.

```
SELECT user_id AS duplicates
FROM user_has_viewed_movie
INTERSECT SELECT user_id FROM user_has_wished_movie;
```



	duplicates
1	4
2	2

#### 1.4.7 LEFT JOIN; RIGHT JOIN; FULL OUTER JOIN; NATURAL JOIN

To select data from the table on the left that may or may not have corresponding rows in the table on the right, you use the LEFT JOIN clause.

```
SELECT movie_release_date, comment_publish FROM movie
LEFT JOIN movie_comments mc
ON movie.movie_id = mc.movie_id;
```

Output (g) LEFT JOIN; RIGHT...ER JOIN; NATURAL JOIN x		
13 rows		
	movie_release_date	comment_publish
1	1994-04-21	2020-01-01 00:00:00.000000
2	1994-04-21	2020-01-01 00:00:00.000000
3	2005-06-29	2020-01-01 00:00:00.000000
4	1998-05-16	2020-01-01 00:00:00.000000
5	1998-05-16	2020-01-01 00:00:00.000000
6	2019-04-30	2020-01-01 00:00:00.000000
7	2019-04-30	2020-01-01 00:00:00.000000
8	2005-06-29	2020-01-01 00:00:00.000000
9	2019-04-30	2020-01-01 00:00:00.000000

The RIGHT JOIN selects all rows from the right table whether or not they have matching rows from the left table.

```
SELECT movie_release_date, comment_publish FROM movie
RIGHT JOIN movie_comments mc
ON movie.movie_id = mc.movie_id;
```

Output Result 37 x		
9 rows		
	movie_release_date	comment_publish
1	1994-04-21	2020-01-01 00:00:00.000000
2	1994-04-21	2020-01-01 00:00:00.000000
3	2005-06-29	2020-01-01 00:00:00.000000
4	1998-05-16	2020-01-01 00:00:00.000000
5	1998-05-16	2020-01-01 00:00:00.000000
6	2019-04-30	2020-01-01 00:00:00.000000
7	2019-04-30	2020-01-01 00:00:00.000000
8	2005-06-29	2020-01-01 00:00:00.000000
9	2019-04-30	2020-01-01 00:00:00.000000

The FULL OUTER JOIN combines the results of both the LEFT JOIN and the RIGHT JOIN.

```
SELECT movie_release_date, comment_publish
FROM movie
FULL OUTER JOIN movie_comments mc
ON movie.movie_id = mc.movie_id;
```

The screenshot shows a database interface with a query result of 13 rows. The columns are 'movie\_release\_date' and 'comment\_publish'. The data is as follows:

	movie_release_date	comment_publish
1	1994-04-21	2020-01-01 00:00:00.000000
2	1994-04-21	2020-01-01 00:00:00.000000
3	2005-06-29	2020-01-01 00:00:00.000000
4	1998-05-16	2020-01-01 00:00:00.000000
5	1998-05-16	2020-01-01 00:00:00.000000
6	2019-04-30	2020-01-01 00:00:00.000000
7	2019-04-30	2020-01-01 00:00:00.000000
8	2005-06-29	2020-01-01 00:00:00.000000
9	2019-04-30	2020-01-01 00:00:00.000000

A NATURAL JOIN is a join that creates an implicit join based on the same column names in the joined tables.

```
SELECT * FROM movie NATURAL JOIN movie_comments;
```

The screenshot shows a database interface with a query result of 9 rows. The columns are 'movie\_id', 'movie\_name', 'movie\_description', 'movie\_release\_date', 'movie\_budget', 'comment\_id', 'user\_id', 'comment\_text', and 'comment\_publish'. The data is as follows:

movie_id	movie_name	movie_description	movie_release_date	movie_budget	comment_id	user_id	comment_text	comment_publish
1	pulp fiction	<null>	1994-04-21	950000.00	1	2	just film	2020-01-01 00:00:00.000000
2	pulp fiction	<null>	1994-04-21	950000.00	2	4	so bad	2020-01-01 00:00:00.000000
3	lady vengeance	<null>	2005-06-29	123456.00	3	10	i liked it	2020-01-01 00:00:00.000000
4	the truman show	<null>	1998-05-16	600000.00	4	7	so boring	2020-01-01 00:00:00.000000
5	the truman show	<null>	1998-05-16	600000.00	5	2	give me back my time	2020-01-01 00:00:00.000000
6	parasite	<null>	2019-04-30	50000.00	6	8	great!	2020-01-01 00:00:00.000000
7	parasite	<null>	2019-04-30	50000.00	7	8	great!	2020-01-01 00:00:00.000000
8	Harry Potter and the Goblet of Fire	best film of this year	2005-06-29	8959294.00	8	12	Amazing!	2020-01-01 00:00:00.000000
9	parasite	<null>	2019-04-30	50000.00	9	8	great!	2020-01-01 00:00:00.000000

## 1.5 Use in one query: LEFT JOIN, GROUP BY, HAVING, ORDER BY, AVG, and DISTINCT

First, we need to add a few corrections to make the following query look better.

```

INSERT INTO
movie(movie_name, movie_description,
movie_release_date, movie_budget)
VALUES ('Harry Potter and the Goblet of Fire',
'best film of this year',
'2005-06-29', 8959294);

INSERT INTO movie_comments(user_id, movie_id,
comment_text, comment_publish)
VALUES (12, 7, 'Amazing!', '2020-01-01');

```

```

postgres.public> INSERT INTO movie(movie_name, movie_description, movie_release_date, movie_budget)
VALUES ('Harry Potter and the Goblet of Fire', 'best film of this year', '2005-06-29', 8959294)
[2023-11-18 20:05:00] 1 row affected in 14 ms
postgres.public> INSERT INTO movie_comments(user_id, movie_id, comment_text, comment_publish)
VALUES (12, 7, 'Amazing!', '2020-01-01')

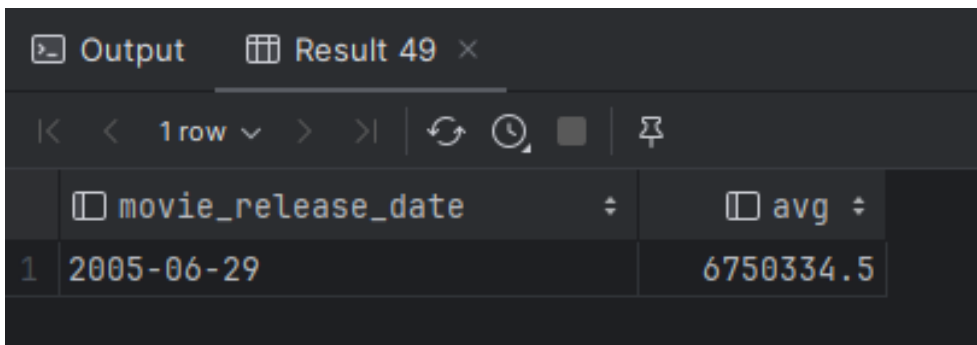
```

This query appears to retrieve information about a specific movie release date ('2005-06-29') from two tables: movie and movie\_comments.

```

SELECT DISTINCT movie_release_date, avg(movie_budget)
FROM movie LEFT JOIN movie_comments mc
ON movie.movie_id = mc.movie_id
GROUP BY movie_release_date
HAVING movie_release_date = '2005-06-29';

```



	movie_release_date	avg
1	2005-06-29	6750334.5

## 1.6 Create a query that will use any aggregate function and GROUP BY with HAVING

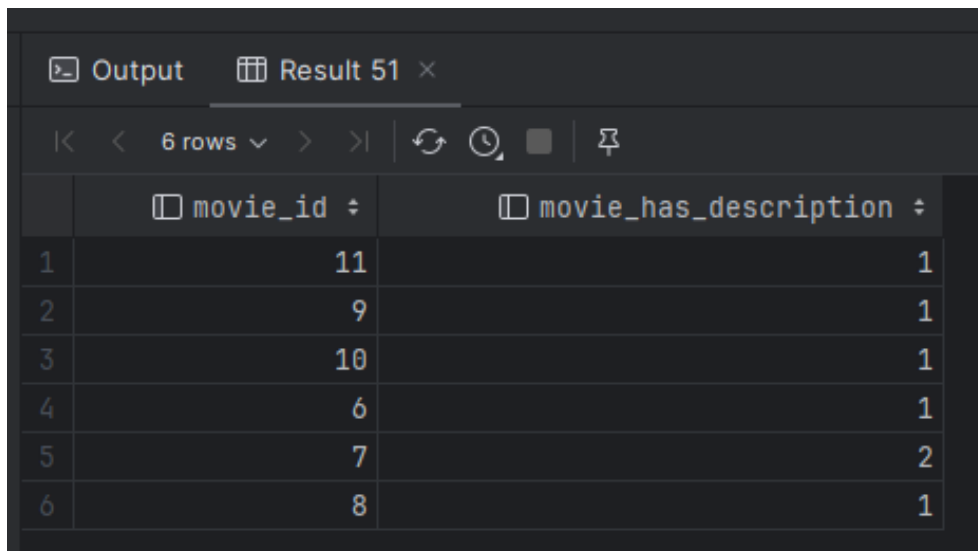
First, we need to add a few corrections to make the following query look better.

```
INSERT INTO
movie(movie_name, movie_description,
movie_release_date, movie_budget)
VALUES ('Oppenheimer', 'robert',
'2023-07-20', 1500000);
```

```
postgres.public> INSERT INTO movie(movie_name, movie_description, movie_release_date, movie_budget)
VALUES ('Oppenheimer', 'robert', '2023-07-20', 1500000)
```

This query is designed to count the number of comments (or occurrences) for each movie that has a description.

```
SELECT movie.movie_id, count(movie.movie_id)
AS movie_has_description FROM movie
LEFT JOIN movie_comments mc
ON movie.movie_id = mc.movie_id
GROUP BY movie.movie_id, movie_description
HAVING movie.movie_description IS NOT NULL;
```



The screenshot shows a database interface with a query result window titled "Result 51". It displays 6 rows of data. The columns are "movie\_id" and "movie\_has\_description". The data is as follows:

	movie_id	movie_has_description
1	11	1
2	9	1
3	10	1
4	6	1
5	7	2
6	8	1

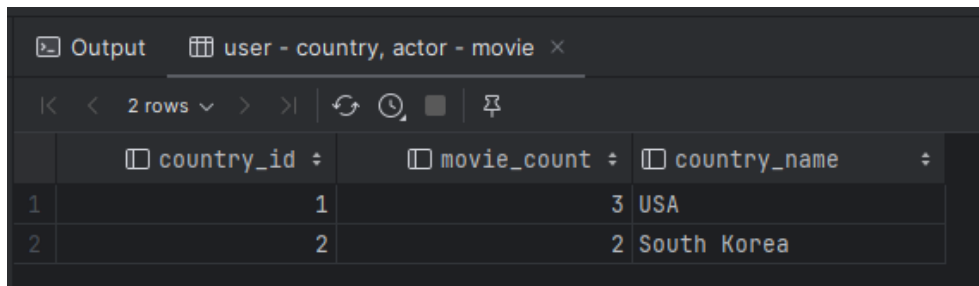
## 1.7 Create a query that will join at least three tables and will use GROUP BY, COUNT, and HAVING

This SQL query is designed to count the number of movies associated with each country that has at least one movie associated with it.

```

SELECT country.country_id, count(movie.movie_id)
AS movie_count, country_name FROM country
LEFT JOIN movie_has_country
ON country.country_id = movie_has_country.country_id
LEFT JOIN movie
ON movie_has_country.movie_id = movie.movie_id
GROUP BY country.country_id,
movie_has_country.country_id
HAVING count(movie.movie_id) > 0;

```



The screenshot shows a database query output window with the title "user - country, actor - movie". It displays 2 rows of data. The columns are country\_id, movie\_count, and country\_name.

	country_id	movie_count	country_name
1	1	3	USA
2	2	2	South Korea

## 1.8 Create a query that will return the data from an arbitrary table for the last one and half days

```

INSERT INTO movie (movie_name, movie_description,
movie_release_date, movie_budget)
VALUES ('BeeMovie', 'bees', '2023-11-17 ', 95433 );

```

```

postgres.public> INSERT INTO movie (movie_name, movie_description, movie_release_date, movie_budget)
VALUES ('BeeMovie', 'bees', '2023-11-17 ', 95433 )

```

This query retrieves movie\_release\_date values from the movie table where the movie\_release\_date is greater than or equal to 36 hours before the current date and time.

```

SELECT movie_release_date FROM movie
WHERE movie.movie_release_date >=
(current_date - INTERVAL '36 hours');

```

Output postgres.public.movie x	
<div> <div>1 row</div> <div> <div>movie_release_date</div> <div>2023-11-17</div> </div> </div>	

### 1.9 Create a query that will return data from the last month (starting from the first day of the month)

```
INSERT INTO movie(movie_name, movie_description,
movie_release_date, movie_budget)
VALUES ('New Film', 'test last month',
'2023-10-09', 5434);
```

```
postgres.public> INSERT INTO movie(movie_name, movie_description, movie_release_date, movie_budget)
VALUES ('New Film', 'test last month', '2023-10-09', 5434)
```

This SQL query retrieves `movie_release_date` values from the `movie` table where the month of the release date is equal to the previous month and the year is equal to the current year.

```
SELECT movie_release_date FROM movie
WHERE date_part('month', movie_release_date) =
date_part('month', current_date) - 1
AND date_part('year', movie_release_date) =
date_part('year', current_date);
```

Output postgres.public.movie	
3 rows	
	movie_release_date
1	2023-10-09
2	2023-10-19
3	2023-10-09

### 1.10 Write a select that will remove accents on a selected string

```
INSERT INTO movie(movie_name, movie_description,
movie_release_date, movie_budget)
VALUES ('Zabijáci rozkvetlého měsíce' ,
'velmi dlouhy film :D', '2023-10-19', 9686575);
```

```
postgres.public> INSERT INTO movie(movie_name, movie_description, movie_release_date, movie_budget)
VALUES ('Zabijáci rozkvetlého měsíce' , 'velmi dlouhy film :D', '2023-10-19', 9686575)
```

The statement `CREATE EXTENSION IF NOT EXISTS unaccent SCHEMA public;` is used to create the `unaccent` extension in the public schema of a PostgreSQL database, but only if it doesn't already exist.

```
CREATE EXTENSION
IF NOT EXISTS unaccent SCHEMA public;
```

```
postgres.public> CREATE EXTENSION IF NOT EXISTS unaccent SCHEMA public
```

```
SELECT unaccent(movie_name)
AS movie , movie_description FROM movie;
```



Output some Czech surname i...se that has accents). x

14 rows

	movie	movie_description
1	Oppenheimer	robert
2	Harry Potter and the Goblet of Fire	best film of this year
3	New Film	test last month
4	Zabijaci rozkvetleho mesice	velmi dlouhy film :D
5	pulp fiction	<null>
6	the dictator	<null>
7	the truman show	<null>
8	parasite	<null>
9	lady vengeance	<null>
10	Harry Potter and the Goblet of Fire	best film of this year
11	Oppenheimer	robert
12	BeeMovie	bees
13	New Film	test last month
14	Zabijaci rozkvetleho mesice	velmi dlouhy film :D

This query retrieves the columns `movie_name` (aliased as `movie`) and `movie_description` from the `movie` table where the lowercased, unaccented version of `movie_name` contains the substring 'zabijaci'.

```
SELECT movie_name AS movie, movie_description
FROM movie
WHERE lower(unaccent(movie_name)) LIKE '%zabijaci%';
```

Output searches the person ...or "Seda" is entered. x

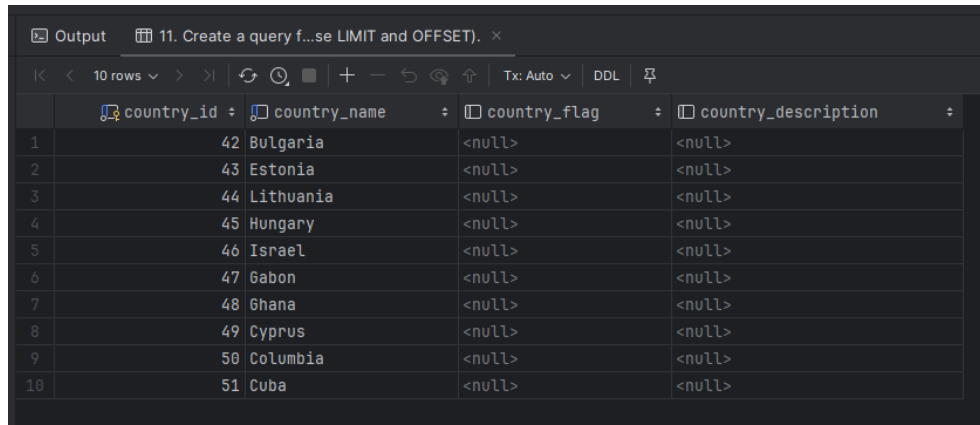
2 rows

	movie	movie_description
1	Zabijáci rozkvetlého měsíce	velmi dlouhy film :D
2	Zabijáci rozkvetlého měsíce	velmi dlouhy film :D

### 1.11 Create a query for pagination in an application (use LIMIT and OFFSET)

This SQL query retrieves all columns from the `country` table, orders the result set by the `country_id` in ascending order, skips the first 40 rows, and then limits the result to 10 rows.

```
SELECT * FROM country
ORDER BY country_id LIMIT 10 OFFSET 40;
```



The screenshot shows a database query output window titled "Output" and "11. Create a query f...se LIMIT and OFFSET).". The window displays 10 rows of data. The columns are: country\_id, country\_name, country\_flag, and country\_description. The data is as follows:

	country_id	country_name	country_flag	country_description
1	42	Bulgaria	<null>	<null>
2	43	Estonia	<null>	<null>
3	44	Lithuania	<null>	<null>
4	45	Hungary	<null>	<null>
5	46	Israel	<null>	<null>
6	47	Gabon	<null>	<null>
7	48	Ghana	<null>	<null>
8	49	Cyprus	<null>	<null>
9	50	Columbia	<null>	<null>
10	51	Cuba	<null>	<null>

## 1.12 Create a query that will use a subquery in FROM

This query retrieves all columns from a derived table named ccl, which is created by selecting the `country_id` column from the `country` table where the `country_id` is greater than 40.

```
SELECT * FROM
(SELECT country_id FROM country
WHERE country_id > 40) AS ccl;
```



Output 13. Create a query t... the WHERE condition. x		
movie_name	movie_release_date	movie_budget
1 Harry Potter and the Goblet of Fire	2005-06-29	8959294.00
2 Zabijáci rozkvetlého měsíce	2023-10-19	9686575.00
3 pulp fiction	1994-04-21	9500000.00
4 Harry Potter and the Goblet of Fire	2005-06-29	8959294.00
5 Zabijáci rozkvetlého měsíce	2023-10-19	9686575.00

## 1.14 Create a query that will join at least five tables

This SQL query performs multiple INNER JOIN operations to combine data from several tables: movie, movie\_comments, movie\_has\_director, director, movie\_has\_actor, and actor. The purpose of the query seems to be to retrieve information about movies along with associated comments, directors, and actors.

```
SELECT * FROM movie
INNER JOIN movie_comments
ON movie.movie_id = movie_comments.movie_id
INNER JOIN movie_has_director
ON movie.movie_id = movie_has_director.movie_id
INNER JOIN director
ON movie_has_director.director_id =
director.director_id
INNER JOIN movie_has_actor
ON movie.movie_id = movie_has_actor.movie_id
INNER JOIN actor
ON movie_has_actor.actor_id = actor.actor_id;
```

Output 14. Create a query t...at least five tables. x								
movie_id	movie_name	movie_description	movie_release_date	movie_budget	comment_id	user_id	movie_comments.movie_id	comment_text
1	pulp fiction	<null>	1994-04-21	9500000.00	1	2	1	just film
2	pulp fiction	<null>	1994-04-21	9500000.00	2	4	1	so bad
3	the truman show	<null>	1998-05-16	6000000.00	4	7	3	so boring
4	the truman show	<null>	1998-05-16	6000000.00	5	2	3	give me back my time
5	parasite	<null>	2019-04-30	50000.00	6	8	4	great!
6	parasite	<null>	2019-04-30	50000.00	7	8	4	great!
7	parasite	<null>	2019-04-30	50000.00	9	8	4	great!
8	lady vengeance	<null>	2005-06-29	123456.00	3	10	5	i liked it
9	lady vengeance	<null>	2005-06-29	123456.00	3	10	5	i liked it

## 1.15 Modify the database from the first project assignment to improve integrity constraints

```
ALTER TABLE user_detail
ADD CONSTRAINT user_detail_pk2
UNIQUE (user_email);

ALTER TABLE country
ADD CONSTRAINT country_pk2
UNIQUE (country_name);
```

#### **1.15.1 Set/improve cascading, explain places where you set/improve cascading and why?**

In simpler terms, this modification ensures that the database maintains consistency by automatically handling the removal of dependent records. For example, if a movie or user is deleted, any associated details in the `user_details` table or comments in the `movie_comments` table will be deleted as well. This is achieved through cascading actions, streamlining the process of maintaining data integrity in your relational database.

```
ALTER TABLE user_detail
ADD CONSTRAINT user_detail_fk
FOREIGN KEY (user_id)
REFERENCES "user" (user_id)
ON DELETE CASCADE;

ALTER TABLE user_has_wished_movie
ADD CONSTRAINT user_has_wished_movie__user_id_fk
FOREIGN KEY (user_id)
REFERENCES "user" (user_id)
ON DELETE CASCADE;

ALTER TABLE movie_comments
ADD CONSTRAINT movie_comments__user_fk
FOREIGN KEY (user_id)
REFERENCES "user" (user_id)
ON DELETE CASCADE;

ALTER TABLE movie_comments
ADD CONSTRAINT movie_comments__movie_fk
FOREIGN KEY (movie_id)
REFERENCES movie (movie_id)
ON DELETE CASCADE;
```

```

ALTER TABLE user_has_viewed_movie
ADD CONSTRAINT user_has_viewed_movie__user_fk
FOREIGN KEY (user_id)
REFERENCES "user" (user_id)
ON DELETE CASCADE;

ALTER TABLE user_has_viewed_movie
ADD CONSTRAINT user_has_viewed_movie__movie_fk
FOREIGN KEY (movie_id)
REFERENCES movie (movie_id)
ON DELETE CASCADE;

ALTER TABLE user_has_favorite_genre
ADD CONSTRAINT user_has_favorite_genre__user_fk
FOREIGN KEY (user_id)
REFERENCES "user" (user_id)
ON DELETE CASCADE;

ALTER TABLE user_has_favorite_genre
ADD CONSTRAINT user_has_favorite_genre__genre_fk
FOREIGN KEY (genre_id)
REFERENCES genre (genre_id)
ON DELETE CASCADE;

ALTER TABLE user_has_favorite_director
ADD CONSTRAINT user_has_favorite_director__user_fk
FOREIGN KEY (user_id)
REFERENCES "user" (user_id)
ON DELETE CASCADE;

ALTER TABLE user_has_favorite_director
ADD CONSTRAINT
user_has_favorite_director__director_fk
FOREIGN KEY (director_id)
REFERENCES director (director_id)
ON DELETE CASCADE;

```

```
ALTER TABLE user_has_favorite_actor
ADD CONSTRAINT user_has_favorite_actor__user_fk
FOREIGN KEY (user_id)
REFERENCES "user" (user_id)
ON DELETE CASCADE;

ALTER TABLE user_has_favorite_actor
ADD CONSTRAINT user_has_favorite_actor__actor_fk
FOREIGN KEY (actor_id)
REFERENCES actor (actor_id)
ON DELETE CASCADE;
```



```

postgres.public> ALTER TABLE movie_comments
                  ADD CONSTRAINT movie_comments_user_id_fk
                  FOREIGN KEY (user_id)
                  REFERENCES "user" (user_id)
                  ON DELETE CASCADE
[2023-11-18 21:53:31] completed in 13 ms
postgres.public> ALTER TABLE movie_comments
                  ADD CONSTRAINT movie_comments_movie_id_fk
                  FOREIGN KEY (movie_id)
                  REFERENCES movie (movie_id)
                  ON DELETE CASCADE
[2023-11-18 21:54:58] completed in 15 ms
postgres.public> ALTER TABLE user_has_viewed_movie
                  ADD CONSTRAINT user_has_viewed_movie_user_id_fk
                  FOREIGN KEY (user_id)
                  REFERENCES "user" (user_id)
                  ON DELETE CASCADE
[2023-11-18 21:55:01] completed in 13 ms
postgres.public> ALTER TABLE user_has_viewed_movie
                  ADD CONSTRAINT user_has_viewed_movie_movie_id_fk
                  FOREIGN KEY (movie_id)
                  REFERENCES movie (movie_id)
                  ON DELETE CASCADE
[2023-11-18 21:55:03] completed in 14 ms
postgres.public> ALTER TABLE user_has_favorite_genre
                  ADD CONSTRAINT user_has_favorite_genre_user_id_fk
                  FOREIGN KEY (user_id)
                  REFERENCES "user" (user_id)
                  ON DELETE CASCADE
[2023-11-18 21:55:06] completed in 15 ms
postgres.public> ALTER TABLE user_has_favorite_genre
                  ADD CONSTRAINT user_has_favorite_genre__fk
                  FOREIGN KEY (genre_id)
                  REFERENCES genre (genre_id)
                  ON DELETE CASCADE
[2023-11-18 21:55:15] completed in 7 ms

```

```

postgres.public> ALTER TABLE user_has_favorite_director
                  ADD CONSTRAINT user_has_favorite_director_user_id_fk
                  FOREIGN KEY (user_id)
                  REFERENCES "user" (user_id)
                  ON DELETE CASCADE
[2023-11-18 21:55:38] completed in 18 ms
postgres.public> ALTER TABLE user_has_favorite_director
                  ADD CONSTRAINT user_has_favorite_director_director_id_fk
                  FOREIGN KEY (director_id)
                  REFERENCES director (director_id)
                  ON DELETE CASCADE
[2023-11-18 21:55:40] completed in 14 ms
postgres.public> ALTER TABLE user_has_favorite_actor
                  ADD CONSTRAINT user_has_favorite_actor_user_id_fk
                  FOREIGN KEY (user_id)
                  REFERENCES "user" (user_id)
                  ON DELETE CASCADE
[2023-11-18 21:55:42] completed in 4 ms
postgres.public> ALTER TABLE user_has_favorite_actor
                  ADD CONSTRAINT user_has_favorite_actor_actor_id_fk
                  FOREIGN KEY (actor_id)
                  REFERENCES actor (actor_id)
                  ON DELETE CASCADE
[2023-11-18 21:55:44] completed in 13 ms

```

## 1.16 Create database indexes

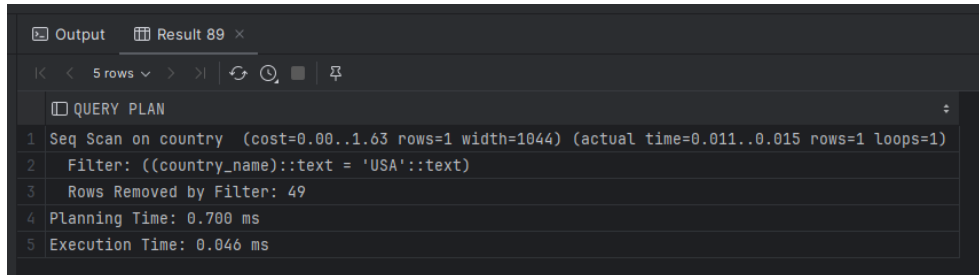
This query is retrieving all columns „\*” from the `user` table where the `user_nickname` column is equal to `'the_key36'`.

```
EXPLAIN ANALYSE SELECT * FROM "user"
WHERE user_nickname = 'the_key36';
```

Output	
Result 88	
5 rows	
QUERY PLAN	
1	Seq Scan on "user" (cost=0.00..1.63 rows=1 width=170) (actual time=0.014..0.015 rows=1 loops=1)
2	Filter: ((user_nickname)::text = 'the_key36'::text)
3	Rows Removed by Filter: 49
4	Planning Time: 0.946 ms
5	Execution Time: 0.027 ms

This query retrieves all columns „\*” from the `country` table where the `country_name` column is equal to `'USA'`.

```
EXPLAIN ANALYSE SELECT * FROM "country"
WHERE country_name = 'USA';
```



QUERY PLAN	
1	Seq Scan on country (cost=0.00..1.63 rows=1 width=1044) (actual time=0.011..0.015 rows=1 loops=1)
2	Filter: ((country_name)::text = 'USA'::text)
3	Rows Removed by Filter: 49
4	Planning Time: 0.700 ms
5	Execution Time: 0.046 ms

### 1.16.1 Explain why it made sense to set indexes for these column(s)

- Adding an index to the `user_nickname` column can be beneficial for read-heavy workloads where quick access to specific user records based on their nickname is essential. It's crucial to analyze the specific query patterns and workload characteristics of your application to determine whether the performance gains from the index outweigh the associated costs.
- Setting an index on the `country_name` column can make sense if the benefits in terms of query performance outweigh the drawbacks in terms of write performance and disk space usage. It's essential to carefully consider the workload and usage patterns of your database when deciding whether to create an index.

### 1.16.2 Explain the benefits and pitfalls of setting up the indexes

#### Benefits

- Faster Sorting and Grouping
- Query Performance Improvement
- Enhanced Unique Constraints

#### Pitfalls

- Write Operation Overhead
- Increased Disk Space Usage
- Over-Indexing

## 1.17 Create arbitrary database procedure

This stored procedure fetches details about a user with a specified ID and either raises a notice with the user's information or raises an exception if the user is not found.

```
CREATE OR REPLACE PROCEDURE
get_user_details(user_id_param INTEGER)
LANGUAGE plpgsql
AS $$
DECLARE
    user_info RECORD;
BEGIN
    SELECT
        *
    INTO
        user_info
    FROM
        "user"
    WHERE
        user_id = user_id_param;
    IF found THEN
        RAISE NOTICE 'User ID: %, Nickname: %',
            user_info.user_id, user_info.user_nickname;
    ELSE
        RAISE EXCEPTION 'User with ID % not found.',
            user_id_param;
    END IF;
END;
$;
```

Here we called this procedure.

```
CALL get_user_details(1);
```

```
postgres.public> CREATE OR REPLACE PROCEDURE get_user_details(user_id_param INTEGER)
    LANGUAGE plpgsql
    AS $$
    DECLARE
        user_info RECORD;
    BEGIN
        SELECT
            *
        INTO
            user_info
        FROM
            "user"
        WHERE
            user_id = user_id_param;
        IF found THEN
            RAISE NOTICE 'User ID: %, Nickname: %',
                user_info.user_id, user_info.user_nickname;
        ELSE
            RAISE EXCEPTION 'User with ID % not found.', user_id_param;
        END IF;
    END;
    $$
[2023-11-18 22:37:05] completed in 13 ms
postgres.public> CALL get_user_details(1)
User ID: 1, Nickname: pabadm
[2023-11-18 22:37:09] completed in 5 ms
```

## 1.18 Create arbitrary database function

This function calculates the average budget of movies released in a specified year and returns the result.

```
CREATE OR REPLACE FUNCTION
calculate_average_budget(year_in INTEGER)
RETURNS DECIMAL(10, 2) AS $$
DECLARE
    total_budget DECIMAL(18, 2) := 0;
    movie_count INTEGER := 0;
    avg_budget DECIMAL(10, 2);
BEGIN
    FOR total_budget, movie_count IN
        SELECT
            COALESCE(SUM(movie_budget), 0),
            COUNT(*)
        FROM
            movie
        WHERE
            EXTRACT(YEAR FROM movie_release_date) =
                year_in
    LOOP
        IF movie_count > 0 THEN
            avg_budget := total_budget / movie_count;
        ELSE
            avg_budget := 0;
        END IF;
    END LOOP;

    RETURN avg_budget;
END;
$$ LANGUAGE plpgsql;
```

```

postgres.public> CREATE OR REPLACE FUNCTION calculate_average_budget(year_in INTEGER)
RETURNS DECIMAL(10, 2) AS $$
DECLARE
    total_budget DECIMAL(18, 2) := 0;
    movie_count INTEGER := 0;
    avg_budget DECIMAL(10, 2);
BEGIN
    FOR total_budget, movie_count IN
        SELECT
            COALESCE(SUM(movie_budget), 0),
            COUNT(*)
        FROM
            movie
        WHERE
            EXTRACT(YEAR FROM movie_release_date) = year_in
    LOOP
        IF movie_count > 0 THEN
            avg_budget := total_budget / movie_count;
        ELSE
            avg_budget := 0;
        END IF;
    END LOOP;

    RETURN avg_budget;
END;
$$ LANGUAGE plpgsql
[2023-11-18 22:30:12] completed in 31 ms
postgres.public> call calculate_average_budget(2023)

```

Here we called this function.

```
SELECT * FROM calculate_average_budget(2023);
```

Output calculate_average_budget x	
<div> <div>1 row</div> <div> <div>calculate_average_budget</div> <div>3211350.14</div> </div> </div>	

## 1.19 Create arbitrary database trigger

In this example, the trigger ensures that if the email is updated in the user table, the corresponding entry in the user\_detail table will also be updated.

```
CREATE OR REPLACE FUNCTION update_user_detail_email()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.user_email IS NOT NULL THEN
        UPDATE user_detail
        SET user_email = NEW.user_email
        WHERE user_id = NEW.user_id;
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER user_update_trigger
AFTER UPDATE ON "user"
FOR EACH ROW
EXECUTE FUNCTION update_user_detail_email();
```

```
postgres.public> CREATE OR REPLACE FUNCTION update_user_detail_email()
                  RETURNS TRIGGER AS $$
                  BEGIN
                      IF NEW.user_email IS NOT NULL THEN
                          UPDATE user_detail
                          SET user_email = NEW.user_email
                          WHERE user_id = NEW.user_id;
                      END IF;

                      RETURN NEW;
                  END;
                  $$ LANGUAGE plpgsql
[2023-11-18 23:00:20] completed in 15 ms
postgres.public> CREATE TRIGGER user_update_trigger
                  AFTER UPDATE ON "user"
                  FOR EACH ROW
                  EXECUTE FUNCTION update_user_detail_email()
[2023-11-18 23:00:24] completed in 12 ms
```



## 1.20 Create arbitrary database view

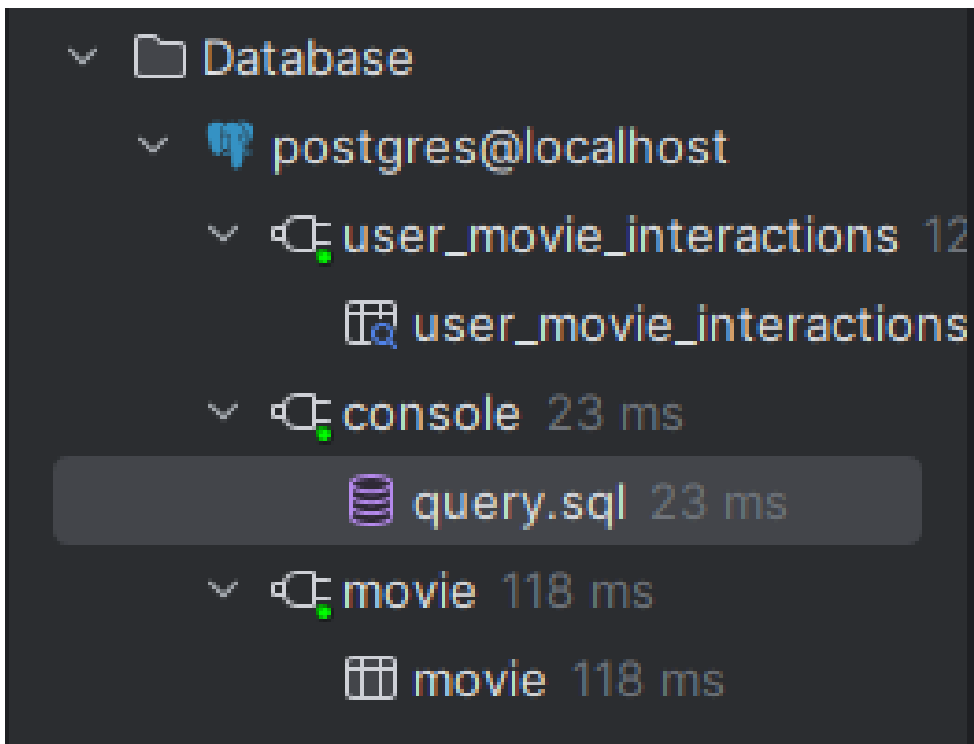
This VIEW provides aggregated information about users and their interactions with movies, including the total number of wished, viewed, and commented movies.

```
CREATE OR REPLACE VIEW user_movie_interactions AS
SELECT
    u.user_id,
    u.user_nickname,
    COUNT(DISTINCT w.movie_id)
    AS total_wished_movies,
    COUNT(DISTINCT mv.movie_id)
    AS total_viewed_movies,
    COUNT(DISTINCT mc.movie_id)
    AS total_commented_movies
FROM
    "user" u
LEFT JOIN
    user_has_wished_movie w ON u.user_id = w.user_id
LEFT JOIN
    user_has_viewed_movie mv
    ON u.user_id = mv.user_id
LEFT JOIN
    movie_comments mc ON u.user_id = mc.user_id
GROUP BY
    u.user_id, u.user_nickname;
```

```

postgres.public> CREATE OR REPLACE VIEW user_movie_interactions AS
SELECT
    u.user_id,
    u.user_nickname,
    COUNT(DISTINCT w.movie_id) AS total_wished_movies,
    COUNT(DISTINCT mv.movie_id) AS total_viewed_movies,
    COUNT(DISTINCT mc.movie_id) AS total_commented_movies
FROM
    "user" u
LEFT JOIN
    user_has_wished_movie w ON u.user_id = w.user_id
LEFT JOIN
    user_has_viewed_movie mv ON u.user_id = mv.user_id
LEFT JOIN
    movie_comments mc ON u.user_id = mc.user_id
GROUP BY
    u.user_id, u.user_nickname
[2023-11-18 23:10:08] completed in 13 ms

```



```
[2023-11-18 23:10:18] Connected
postgres.public> SELECT t.*
                    FROM public.user_movie_interactions t
                    LIMIT 501
[2023-11-18 23:10:18] 50 rows retrieved starting from 1 in 42 ms (execution: 12 ms, fetching: 30 ms)
```

	user_id	user_nickname	total_wished_movies	total_viewed_movies	total_commented_movies
1	1	pabadn	0	0	0
2	2	nuah1946	1	1	2
3	3	piccolo219	1	0	0
4	4	_strawberry_lover_	1	1	1
5	5	guardian	1	0	0
6	6	the_key	0	0	0
7	7	user	1	0	1
8	8	pizzza_guy	0	1	1
9	9	the_boxer	0	0	0
10	10	surname_name	0	2	1
11	11	the_key2	0	0	0
12	12	the_key3	0	0	1
13	13	the_key4	0	0	0
14	14	the_key5	0	0	0
15	15	the_key6	0	0	0
16	16	the_key7	0	0	0
17	17	the_key8	0	0	0
18	18	the_key9	0	0	0
19	19	the_key10	0	0	0
20	20	the_key11	0	0	0
21	21	the_key12	0	0	0
22	22	the_key13	0	0	0
23	23	the_key14	0	0	0
24	24	the_key15	0	0	0
25	25	the_key16	0	0	0
26	26	the_key17	0	0	0
27	27	the_key18	0	0	0
--	--	--	-	-	-

1.21 Create database materialized view (consider some complicated SQL query with several joins, aggregate function, GROUP BY with HAVING and complex WHERE condition). Explain why this materialized view is beneficial/needed.

A materialized view that shows the total number of movies, the average budget, and the number of users who have wished and viewed each movie, but only for movies released after a certain date and with a budget higher than a specified amount.

```

CREATE MATERIALIZED VIEW movie_stats AS
SELECT
    m.movie_id,
    m.movie_name,
    m.movie_release_date,
    COUNT(DISTINCT uwm.user_id) AS num_wished_users,
    COUNT(DISTINCT uvm.user_id) AS num_viewed_users,
    COUNT(DISTINCT uwm.user_id) +
    COUNT(DISTINCT uvm.user_id)
    AS total_interactions,
    AVG(m.movie_budget) AS avg_budget,
    COUNT(*) AS total_movies
FROM
    movie m
LEFT JOIN
    user_has_wished_movie uwm ON m.movie_id =
    uwm.movie_id
LEFT JOIN
    user_has_viewed_movie uvm ON m.movie_id =
    uvm.movie_id
WHERE
    m.movie_budget > 1000
GROUP BY
    m.movie_id, m.movie_name, m.movie_release_date
HAVING
    m.movie_release_date > '1900-01-01';

```

```

CREATE MATERIALIZED VIEW movie_stats AS
SELECT
    m.movie_id,
    m.movie_name,
    m.movie_release_date,
    COUNT(DISTINCT uwm.user_id) AS num_wished_users,
    COUNT(DISTINCT uvm.user_id) AS num_viewed_users,
    COUNT(DISTINCT uwm.user_id) + COUNT(DISTINCT uvm.user_id) AS total_interactions,
    AVG(m.movie_budget) AS avg_budget,
    COUNT(*) AS total_movies
FROM
    movie m
LEFT JOIN
    user_has_wished_movie uwm ON m.movie_id = uwm.movie_id
LEFT JOIN
    user_has_viewed_movie uvm ON m.movie_id = uvm.movie_id
WHERE
    m.movie_budget > 1000
GROUP BY
    m.movie_id, m.movie_name, m.movie_release_date
HAVING
    m.movie_release_date > '1900-01-01';

```

```

postgres.public> CREATE MATERIALIZED VIEW movie_stats AS
SELECT
    m.movie_id,
    m.movie_name,
    m.movie_release_date,
    COUNT(DISTINCT uwm.user_id) AS num_wished_users,
    COUNT(DISTINCT uvm.user_id) AS num_viewed_users,
    COUNT(DISTINCT uwm.user_id) + COUNT(DISTINCT uvm.user_id) AS total_interactions,
    AVG(m.movie_budget) AS avg_budget,
    COUNT(*) AS total_movies
FROM
    movie m
LEFT JOIN
    user_has_wished_movie uwm ON m.movie_id = uwm.movie_id
LEFT JOIN
    user_has_viewed_movie uvm ON m.movie_id = uvm.movie_id
WHERE
    m.movie_budget > 1000
GROUP BY
    m.movie_id, m.movie_name, m.movie_release_date
HAVING
    m.movie_release_date > '1900-01-01'
[2023-11-18 23:21:37] 13 rows affected in 7 ms

```

	movie_id	movie_name	movie_release_date	num_wished_users	num_viewed_users	total_interactions	avg_budget	total_movies
1	1	pulp fiction	1994-04-21	2	0	2	9500000	2
2	3	the truman show	1998-05-16	1	2	3	6000000	2
3	4	parasite	2019-04-30	0	1	1	500000	1
4	5	lady vengeance	2005-06-29	1	1	2	123456	1
5	6	Oppenheimer	2023-07-20	0	0	0	15000000	1
6	7	Harry Potter and the Goblet of Fire	2005-06-29	0	0	0	8959294	1
7	8	New Film	2023-10-09	0	0	0	5434	1
8	9	Zabijáci rozkvetlého měsíce	2023-10-19	0	0	0	9686575	1
9	10	Harry Potter and the Goblet of Fire	2005-06-29	0	0	0	8959294	1
10	11	Oppenheimer	2023-07-20	0	0	0	15000000	1
11	12	BeeMovie	2023-11-17	0	0	0	95433	1
12	13	New Film	2023-10-09	0	0	0	5434	1
13	14	Zabijáci rozkvetlého měsíce	2023-10-19	0	0	0	9686575	1

```

[2023-11-18 23:21:44] Connected
postgres.public> SELECT t.*
                  FROM public.movie_stats t
                  LIMIT 501
[2023-11-18 23:21:44] 13 rows retrieved starting from 1 in 49 ms (execution: 3 ms, fetching: 46 ms)

```

## Benefits

- Improved Query Performance
- Reduced Query Complexity
- Faster Response Times for Reporting
- Consistency and Data Integrity
- Offline Analysis and Reporting

Keep in mind that materialized views do come with trade-offs, such as the need to manage refresh schedules to keep the data up-to-date and potential storage considerations.

## 2 Part Secutiry

### 2.1 If you use public schema create a new schema for your database called bds

#### 2.1.1 Migrate all your data from public to bds schema

```
CREATE SCHEMA bds;

ALTER TABLE public."user" SET SCHEMA bds;
ALTER TABLE public.movie SET SCHEMA bds;
ALTER TABLE public.user_has_wished_movie
SET SCHEMA bds;
ALTER TABLE public.movie_comments SET SCHEMA bds;
ALTER TABLE public.user_has_viewed_movie
SET SCHEMA bds;
ALTER TABLE public.user_detail SET SCHEMA bds;
ALTER TABLE public.genre SET SCHEMA bds;
ALTER TABLE public.country SET SCHEMA bds;
ALTER TABLE public.director SET SCHEMA bds;
ALTER TABLE public.actor SET SCHEMA bds;
ALTER TABLE public.user_has_favorite_genre
SET SCHEMA bds;
ALTER TABLE public.user_has_favorite_director
SET SCHEMA bds;
ALTER TABLE public.user_has_favorite_actor
SET SCHEMA bds;
ALTER TABLE public.movie_has_country
SET SCHEMA bds;
ALTER TABLE public.movie_has_genre
SET SCHEMA bds;
ALTER TABLE public.movie_has_director
SET SCHEMA bds;
ALTER TABLE public.movie_has_actor
SET SCHEMA bds;
```

```

postgres.public> CREATE SCHEMA bds
[2023-11-18 23:36:36] completed in 13 ms
postgres.public> ALTER TABLE public."user" SET SCHEMA bds
[2023-11-18 23:36:40] completed in 12 ms
postgres.public> ALTER TABLE public.movie SET SCHEMA bds
[2023-11-18 23:36:42] completed in 13 ms
postgres.public> ALTER TABLE public.user_has_wished_movie SET SCHEMA bds
[2023-11-18 23:36:44] completed in 12 ms
postgres.public> ALTER TABLE public.movie_comments SET SCHEMA bds
[2023-11-18 23:36:45] completed in 12 ms
postgres.public> ALTER TABLE public.user_has_viewed_movie SET SCHEMA bds
[2023-11-18 23:36:47] completed in 3 ms
postgres.public> ALTER TABLE public.user_detail SET SCHEMA bds
[2023-11-18 23:36:48] completed in 13 ms
postgres.public> ALTER TABLE public.genre SET SCHEMA bds
[2023-11-18 23:36:49] completed in 12 ms
postgres.public> ALTER TABLE public.country SET SCHEMA bds
[2023-11-18 23:36:52] completed in 12 ms
postgres.public> ALTER TABLE public.director SET SCHEMA bds
[2023-11-18 23:36:54] completed in 12 ms
postgres.public> ALTER TABLE public.actor SET SCHEMA bds
[2023-11-18 23:36:58] completed in 13 ms
postgres.public> ALTER TABLE public.user_has_favorite_genre SET SCHEMA bds
[2023-11-18 23:37:00] completed in 12 ms
postgres.public> ALTER TABLE public.user_has_favorite_director SET SCHEMA bds
[2023-11-18 23:37:02] completed in 13 ms
postgres.public> ALTER TABLE public.user_has_favorite_actor SET SCHEMA bds
[2023-11-18 23:37:03] completed in 12 ms
postgres.public> ALTER TABLE public.movie_has_country SET SCHEMA bds
[2023-11-18 23:37:05] completed in 12 ms
postgres.public> ALTER TABLE public.movie_has_genre SET SCHEMA bds
[2023-11-18 23:37:06] completed in 12 ms
postgres.public> ALTER TABLE public.movie_has_director SET SCHEMA bds
[2023-11-18 23:37:08] completed in 13 ms
postgres.public> ALTER TABLE public.movie_has_actor SET SCHEMA bds
[2023-11-18 23:37:10] completed in 12 ms
postgres.public> REVOKE CREATE ON SCHEMA public FROM PUBLIC
[2023-11-18 23:37:13] completed in 13 ms

```

### 2.1.2 Revoke create on schema public from public

After executing this REVOKE statement, users will need explicit permissions to create objects within the public schema.

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```



### 2.1.3 Explain why it might be useful to avoid public schema

- Security
- Organization
- Maintenance
- Schema Naming Conflicts
- Enhanced Security Practices
- Data Isolation

## 2.2 Create two roles bds-app and bds-script in your database

These queries set up roles, define permissions on tables and sequences, enable row-level security, and create a row-level security policy.

```
CREATE ROLE bds_app;
CREATE ROLE bds_script;

GRANT SELECT, INSERT, UPDATE,
DELETE ON TABLE bds."user" TO bds_app;

ALTER TABLE bds."user" ENABLE ROW LEVEL SECURITY;

CREATE POLICY restrict_fields
  ON bds."user"
  USING (NOT ('user_password'::text =
    current_setting('app.user_role')::text));

GRANT ALL PRIVILEGES ON ALL SEQUENCES IN SCHEMA bds
TO bds_app;

GRANT SELECT ON TABLE bds.movie TO bds_script;
GRANT SELECT ON TABLE bds.movie_comments
TO bds_script;
```

```

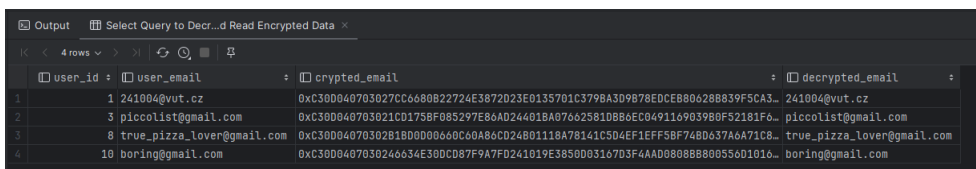
postgres.public> CREATE ROLE bds_app
[2023-11-18 23:50:34] completed in 14 ms
postgres.public> CREATE ROLE bds_script
[2023-11-18 23:50:36] completed in 11 ms
postgres.public> GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE bds."user" TO bds_app
[2023-11-18 23:50:38] completed in 17 ms
postgres.public> ALTER TABLE bds."user" ENABLE ROW LEVEL SECURITY
[2023-11-18 23:50:39] completed in 2 ms
postgres.public> CREATE POLICY restrict_fields
ON bds."User"
USING (NOT ('user_password'::text = current_setting('app.user_role')::text))
[2023-11-18 23:50:41] completed in 14 ms
postgres.public> GRANT ALL PRIVILEGES ON ALL SEQUENCES IN SCHEMA bds TO bds_app
[2023-11-18 23:50:44] completed in 4 ms
postgres.public> GRANT SELECT ON TABLE bds.movie TO bds_script
[2023-11-18 23:50:45] completed in 11 ms
postgres.public> GRANT SELECT ON TABLE bds.movie_comments TO bds_script
[2023-11-18 23:50:47] completed in 11 ms

```

## 2.3 Encrypt selected database columns (using pgcrypto module)

### 2.3.1 Explain why you encrypted these columns

Encrypting columns containing sensitive information like email addresses is a fundamental security practice that safeguards the confidentiality, integrity, and trustworthiness of the data, aligning with best practices and regulatory requirements.



The screenshot shows a database query result with 4 rows. The columns are user\_id, user\_email, encrypted\_email, and decrypted\_email. The data shows that the encrypted\_email column contains long hexadecimal strings, and the decrypted\_email column contains the original email addresses.

user_id	user_email	encrypted_email	decrypted_email
1	241004@vut.cz	0xc30d040703027cc6680b22724e3872023e0135701c379ba3d9b78e0ce880628839f5ca3...	241004@vut.cz
2	3 piccolist@gmail.com	0xc30d040703021cd175bf085297e86ad24401ba0766258108b6ec049116903980f52181f6...	piccolist@gmail.com
3	8 true_pizza_lover@gmail.com	0xc30d0407030281800d00660c60a86cd24b0118a78141c504ef1eff58f748d637a6a71cb...	true_pizza_lover@gmail.com
4	10 boring@gmail.com	0xc30d0407030246634e380cd87f9a7f0241019e3850d0316703f4aad0808888055601016...	boring@gmail.com

```

CREATE EXTENSION IF NOT EXISTS pgcrypto SCHEMA bds;

CREATE TABLE IF NOT EXISTS bds.email_keys (
    id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES
    bds.user_detail(user_id) UNIQUE,
    key TEXT NOT NULL
);

ALTER TABLE bds.user_detail
ADD COLUMN IF NOT EXISTS crypted_email BYTEA;

WITH new_keys AS (
    INSERT INTO bds.email_keys (user_id, key)
    SELECT
        user_id,
        gen_random_bytes(32) AS key
    FROM bds.user_detail WHERE user_email IS NOT NULL
    RETURNING user_id, key
)
UPDATE bds.user_detail AS ud
SET crypted_email =
pgp_sym_encrypt(ud.user_email, nk.key)
FROM new_keys nk
WHERE ud.user_id = nk.user_id;

SELECT
    ud.user_id,
    ud.user_email,
    ud.crypted_email,
    pgp_sym_decrypt(ud.crypted_email, ek.key)
    AS decrypted_email
FROM
    bds.user_detail ud
JOIN
    bds.email_keys ek ON ud.user_id = ek.user_id;

```

## 2.4 Securing the connections to your database instance

### 2.4.1 Explain the purpose of pg\_hba.conf file in PostgreSQL

The `pg_hba.conf` (Host-Based Authentication) file in PostgreSQL is a configuration file that controls client authentication to the PostgreSQL server. It specifies the rules that determine which hosts are allowed to connect to the PostgreSQL server and what authentication methods are used for those connections. The primary purpose of `pg_hba.conf` is to define the security policies for client access.

### 2.4.2 Add an example of how to restrict the connection

```
# TYPE DATABASE USER ADDRESS METHOD
# "local" is for Unix domain socket connections only
local all all trust
# IPv4 local connections:
host all all 127.0.0.1/32 scram-sha-256
# IPv6 local connections:
host all all ::1/128 scram-sha-256
# Allow replication connections from localhost, by a user with the
# replication privilege.
local replication all scram-sha-256
host replication all 127.0.0.1/32 scram-sha-256
host replication all ::1/128 scram-sha-256
host all bds-app 127.0.0.1/32 md5
host all bds-app 127.0.0.1/32 md5
host all bds-script 127.0.0.1/32 md5
```

## 2.5 Specify the security hardening parameters (in production consider/add additional ones). For each of these parameters explain its purpose and possible benefits

```

ALTER SYSTEM SET logging_collector = ON;

ALTER SYSTEM SET log_filename =
'postgresql-%Y-%m-%d %H%M%S.log';

ALTER SYSTEM SET log_file_mode = '0640';

ALTER SYSTEM SET log_truncate_on_rotation = OFF;

ALTER SYSTEM SET log_min_messages = WARNING;

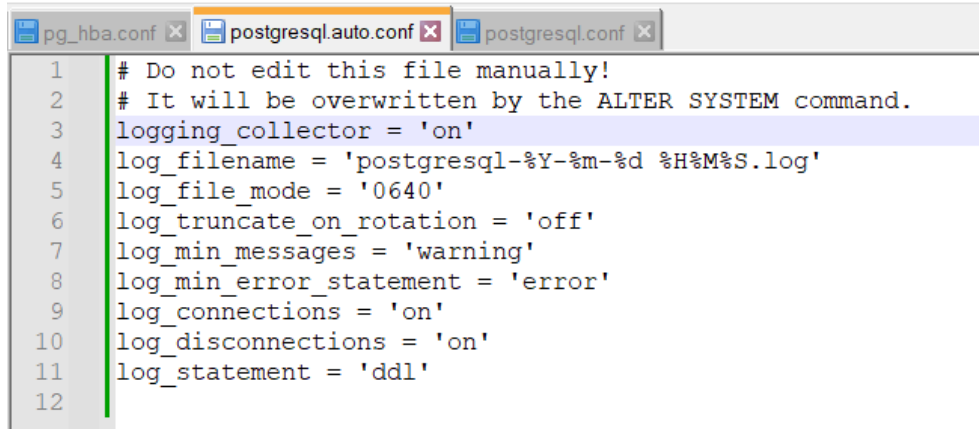
ALTER SYSTEM SET log_min_error_statement = ERROR;

ALTER SYSTEM SET log_connections = ON;

ALTER SYSTEM SET log_disconnections = ON;

ALTER SYSTEM SET log_statement = 'ddl';

```



```

1  # Do not edit this file manually!
2  # It will be overwritten by the ALTER SYSTEM command.
3  logging_collector = 'on'
4  log_filename = 'postgresql-%Y-%m-%d %H%M%S.log'
5  log_file_mode = '0640'
6  log_truncate_on_rotation = 'off'
7  log_min_messages = 'warning'
8  log_min_error_statement = 'error'
9  log_connections = 'on'
10 log_disconnections = 'on'
11 log_statement = 'ddl'
12

```

## 1. log destination = 'stderr'

- (a) **Purpose:** Specifies the destination for log output. In this case, logs will be sent to the standard error output (stderr).
- (b) **Benefits:** Sending logs to stderr allows them to be captured by the system's logging facilities, making them accessible for monitoring and analysis.

## 2. log collector = on

- (a) **Purpose:** Enables the log collector process, which is responsible for capturing log messages generated by the PostgreSQL server.
  - (b) **Benefits:** Enabling the log collector is necessary for centralized log management. It allows logs to be collected and stored in files or forwarded to external logging systems.
3. **log filename = 'postgresql -%Y-%m-%d %H%M%S.log'**
- (a) **Purpose:** Sets the format for the log file names. The specified format includes a timestamp.
  - (b) **Benefits:** Creating log files with timestamps helps in organizing logs, making it easier to identify and analyze issues over time.
4. **log file mode = 0640**
- (a) **Purpose:** Sets the file permissions for the log files.
  - (b) **Benefits:** Specifying appropriate file permissions ensures that the log files are secure and accessible only to authorized users.
5. **log truncate on rotation = off**
- (a) **Purpose:** Determines whether to truncate the log file on rotation. If set to off, the log file will not be truncated.
  - (b) **Benefits:** Keeping log history across rotations can be useful for historical analysis and troubleshooting.
6. **log min messages = warning**
- (a) **Purpose:** Sets the minimum message level to be logged. Messages with a severity level equal to or higher than this setting will be logged.
  - (b) **Benefits:** Filtering out less severe messages helps in focusing on significant events and reduces the volume of log data.
7. **log min error statement = error**
- (a) **Purpose:** Sets the minimum statement level for statements that cause an error to be logged.
  - (b) **Benefits:** Logging only error statements helps in identifying and addressing critical issues without being overwhelmed by routine statements.

#### 8. `log connections = on`

- (a) **Purpose:** Enables logging of successful connection attempts.
- (b) **Benefits:** Logging connections can be valuable for monitoring user activity and identifying potential security issues.

#### 9. `log disconnections = on`

- (a) **Purpose:** Enables logging of disconnection events.
- (b) **Benefits:** Logging disconnections provides insights into user sessions and helps in understanding the lifecycle of connections.

#### 10. `log statement = 'ddl'`

- (a) **Purpose:** Sets the type of statements to be logged. In this case, only DDL (Data Definition Language) statements will be logged.
- (b) **Benefits:** Focusing on DDL statements helps in tracking changes to the database schema, facilitating auditing and debugging.

### 2.5.1 Explain how the SSL can be enabled and configured

```
ssl = on
ssl_ca_file = 'root.crt'
ssl_cert_file = 'server.crt'
ssl_crl_file = ''
ssl_key_file = 'server.key'
ssl_ciphers = 'HIGH:MEDIUM:+3DES:!aNULL'
ssl_prefer_server_ciphers = on
```

#### 1. `ssl = on`

- (a) **Purpose:** Enables SSL support for PostgreSQL.
- (b) **Configuration:** This line alone enables SSL, but the other parameters are necessary for a secure SSL setup.

#### 2. `ssl_ca_file = 'root.crt'`

- (a) **Purpose:** Specifies the location of the Certificate Authority (CA) file. The CA file contains the root certificate used to verify the authenticity of the server's SSL certificate.

- (b) **Configuration:** Provide the path to the CA file, which is used to verify the server's certificate during the SSL handshake.
3. `ssl_cert_file = 'server.crt'`
- (a) **Purpose:** Specifies the location of the server's SSL certificate file.
  - (b) **Configuration:** Provide the path to the server's SSL certificate, which is presented to clients during the SSL handshake.
4. `ssl_crl_file = ''`
- (a) **Purpose:** Specifies the location of the Certificate Revocation List (CRL) file. The CRL is used to check if a certificate has been revoked.
  - (b) **Configuration:** If you have a CRL, provide the path to the CRL file. In your case, it's an empty string, indicating that no CRL is used.
5. `ssl_key_file = 'server.key'`
- (a) **Purpose:** Specifies the location of the server's private key file.
  - (b) **Configuration:** Provide the path to the server's private key, which is used for decrypting data during the SSL handshake.
6. `ssl_ciphers = 'HIGH MEDIUM +3DES !aNULL'`
- (a) **Purpose:** Specifies the allowed SSL ciphers. In this case, it allows ciphers that are considered secure (HIGH and MEDIUM) but excludes 3DES and null ciphers.
  - (b) **Configuration:** Define a list of ciphers that the server can use during the SSL negotiation.
7. `ssl_prefer_server_ciphers = on`
- (a) **Purpose:** Indicates whether the server should prefer the order of ciphers specified in `ssl_ciphers`.
  - (b) **Configuration:** Setting this to on means that the server's order of ciphers will be preferred during the SSL handshake.



## 2.6 Set the data folder

It can be changed in postgresql.conf. data\_directory can be specified.

```
# The default values of these variables are driven from the -D command-line
# option or PGDATA environment variable, represented here as ConfigDir.
data_directory = /data/postgres # use data in another directory
# (change requires restart)
```

```
SHOW data_directory;
```

## 2.7 Optimizing database performance

```
# DB Version: 15
# OS Type: linux
# DB Type: web
# Total Memory (RAM): 32 GB
# CPUs num: 4
# Connections num: 150
# Data Storage: hdd

max_connections = 150
shared_buffers = 8GB
effective_cache_size = 24GB
maintenance_work_mem = 2GB
checkpoint_completion_target = 0.9
wal_buffers = 16MB
default_statistics_target = 100
random_page_cost = 4
effective_io_concurrency = 2
work_mem = 27962kB
huge_pages = try
min_wal_size = 1GB
max_wal_size = 4GB
max_worker_processes = 4
max_parallel_workers_per_gather = 2
max_parallel_workers = 4
max_parallel_maintenance_workers = 2
```

```
-- DB Version: 15
-- OS Type: linux
-- DB Type: web
-- Total Memory (RAM): 32 GB
-- CPUs num: 4
-- Connections num: 150
-- Data Storage: hdd

ALTER SYSTEM SET
    max_connections = '150';
ALTER SYSTEM SET
    shared_buffers = '8GB';
ALTER SYSTEM SET
    effective_cache_size = '24GB';
ALTER SYSTEM SET
    maintenance_work_mem = '2GB';
ALTER SYSTEM SET
    checkpoint_completion_target = '0.9';
ALTER SYSTEM SET
    wal_buffers = '16MB';
ALTER SYSTEM SET
    default_statistics_target = '100';
ALTER SYSTEM SET
    random_page_cost = '4';
ALTER SYSTEM SET
    effective_io_concurrency = '2';
ALTER SYSTEM SET
    work_mem = '27962kB';
ALTER SYSTEM SET
    huge_pages = 'try';
ALTER SYSTEM SET
    min_wal_size = '1GB';
ALTER SYSTEM SET
    max_wal_size = '4GB';
ALTER SYSTEM SET
    max_worker_processes = '4';
ALTER SYSTEM SET
    max_parallel_workers_per_gather = '2';
ALTER SYSTEM SET
    max_parallel_workers = '4';
```

```
ALTER SYSTEM SET  
max_parallel_maintenance_workers = '2';
```

