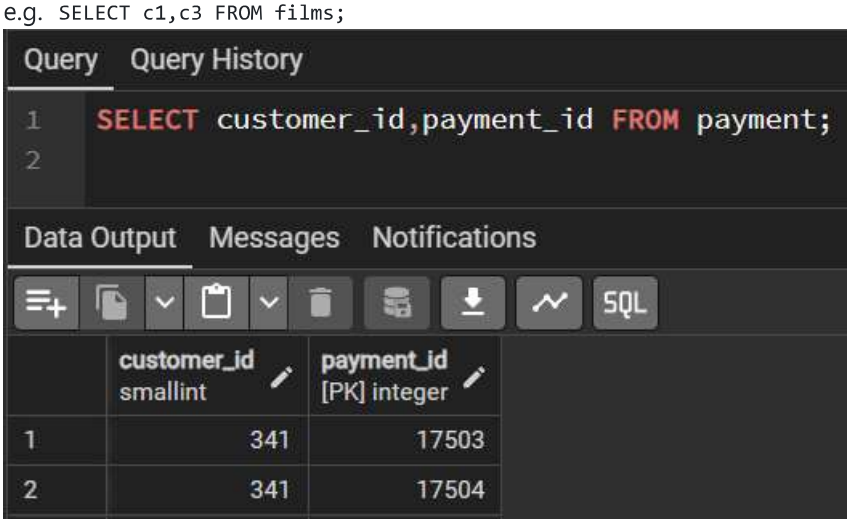
SQL Fundamental Statment

1. "SELECT" command

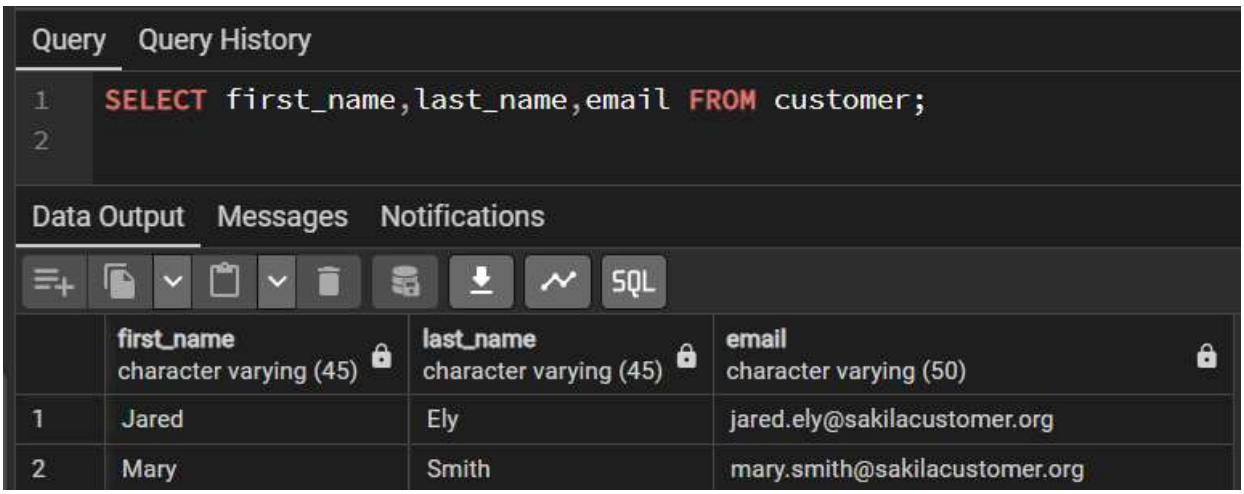
- Syntax: `SELECT <column_name> FROM <Table_Name>`
  - Note: you can add multiple columns names in it separated by Commas
  - e.g. `SELECT c1,c3 FROM films;`



- Using '*' in place of coulmn name will select all columns

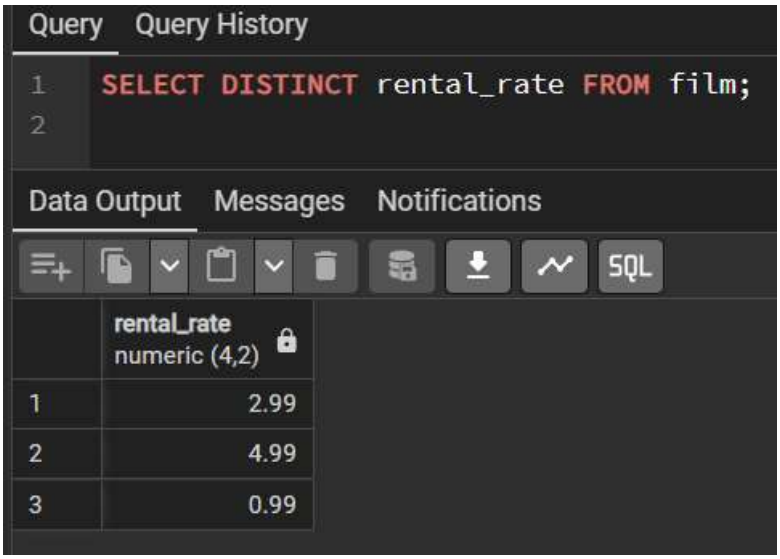
⚔️ Challenge "SELECT"

- Ques) We want to send a promotional email to our exisiting customers. Select first and last names of every customer and theri email address
  - **Solution)** `SELECT first_name,last_name,email FROM customer;`



2. "SELECT DISTINCT" command

- The SELECT DISTINCT command in SQL is used to retrieve unique values from a specific column or a combination of columns in a table. It eliminates duplicate records from the result set.
- Syntax: `SELECT DISTINCT <Coulumn_name> FROM <Table_name>`

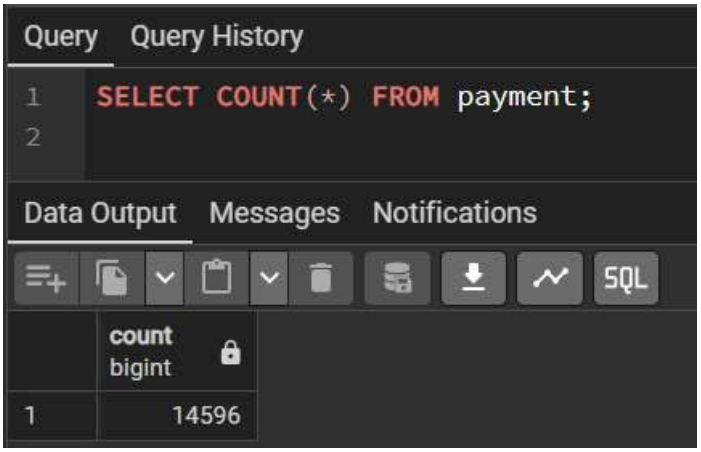


⚔️ Challenge "SELECT DISTINCT"

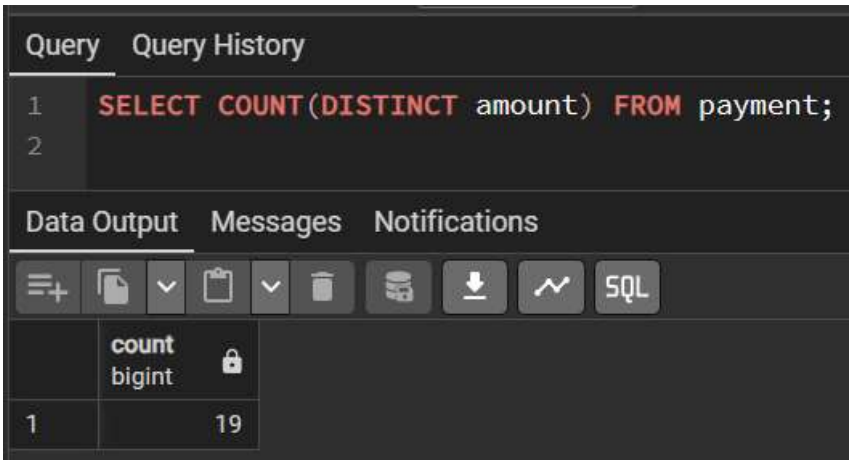
- Ques)We want to know the types of movie ratings we have in our database?
  - **Solution)** `SELECT DISTINCT rating FROM film;`

3. "Count"

- Returns the number of input rows that matches a specific Query Condition
- Syntax: `SELECT COUNT(name) FROM table`



- Note: "()" is must while using COUNT command

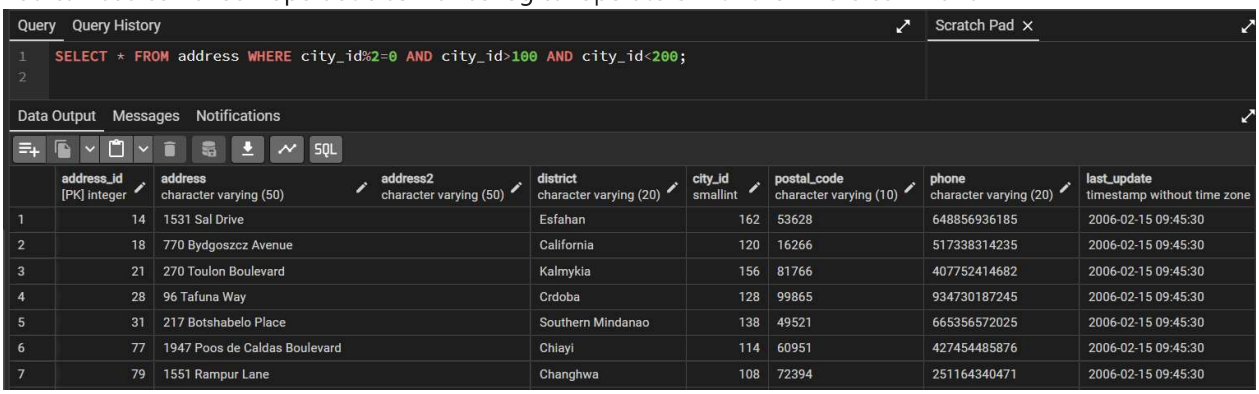


- One specific use case of COUNT is to ***count Non-NULL Values in a Specific Column***
  - `SELECT COUNT(column_name) FROM table` Will return the count of Non-NULL

4. "SELECT WHERE" command

- The SELECT WHERE command in SQL is used to retrieve specific data from a database based on a condition. The WHERE clause filters the records returned by the SELECT statement, so that only rows meeting the specified condition are included in the result.

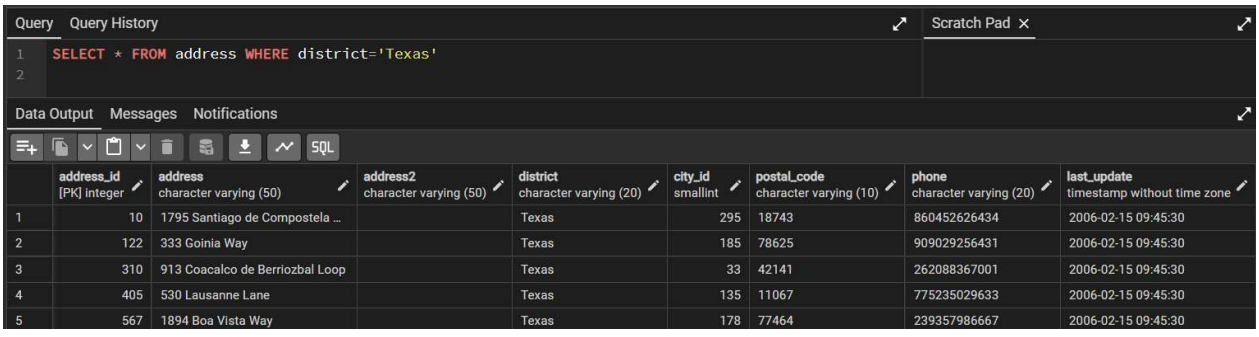
- You can use comarison operaotrs as well as logical operators with the where command



```
SELECT * FROM address WHERE city_id%2=0 AND city_id>100 AND city_id<200;
```

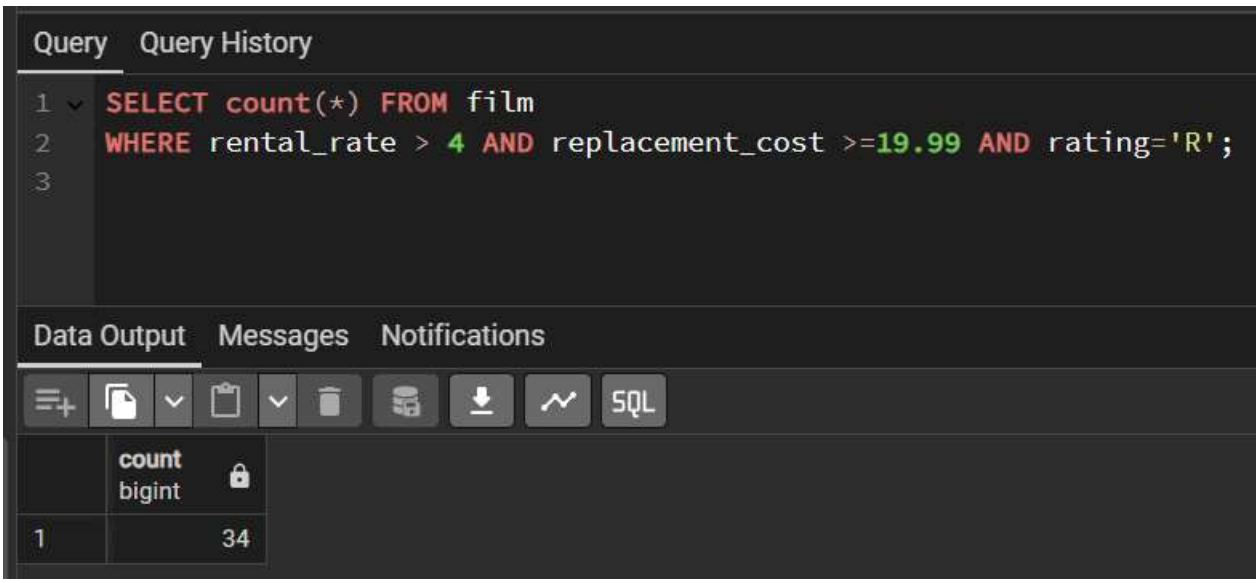
	address_id [PK] integer	address character varying (50)	address2 character varying (50)	district character varying (20)	city_id smallint	postal_code character varying (10)	phone character varying (20)	last_update timestamp without time zone
1	14	1531 Sal Drive		Esfahan	162	53628	648856936185	2006-02-15 09:45:30
2	18	770 Bydgoszcz Avenue		California	120	16266	517338314235	2006-02-15 09:45:30
3	21	270 Toulon Boulevard		Kalmykia	156	81766	407752414682	2006-02-15 09:45:30
4	28	96 Tafuna Way		Crdoba	128	99865	934730187245	2006-02-15 09:45:30
5	31	217 Botshabelo Place		Southern Mindanao	138	49521	665356572025	2006-02-15 09:45:30
6	77	1947 Poos de Caldas Boulevard		Chiayi	114	60951	427454485876	2006-02-15 09:45:30
7	79	1551 Rampur Lane		Changhwa	108	72394	251164340471	2006-02-15 09:45:30

  - **Note: In SQL, the equality operator is a single = instead of ==, which is used in many programming languages like C, C++, and Java.**



```
SELECT * FROM address WHERE district='Texas'
```

	address_id [PK] integer	address character varying (50)	address2 character varying (50)	district character varying (20)	city_id smallint	postal_code character varying (10)	phone character varying (20)	last_update timestamp without time zone
1	10	1795 Santiago de Compostela ...		Texas	295	18743	860452626434	2006-02-15 09:45:30
2	122	333 Goinia Way		Texas	185	78625	909029256431	2006-02-15 09:45:30
3	310	913 Coacalco de Berriozbal Loop		Texas	33	42141	262088367001	2006-02-15 09:45:30
4	405	530 Lausanne Lane		Texas	135	11067	775235029633	2006-02-15 09:45:30
5	567	1894 Boa Vista Way		Texas	178	77464	239357986667	2006-02-15 09:45:30



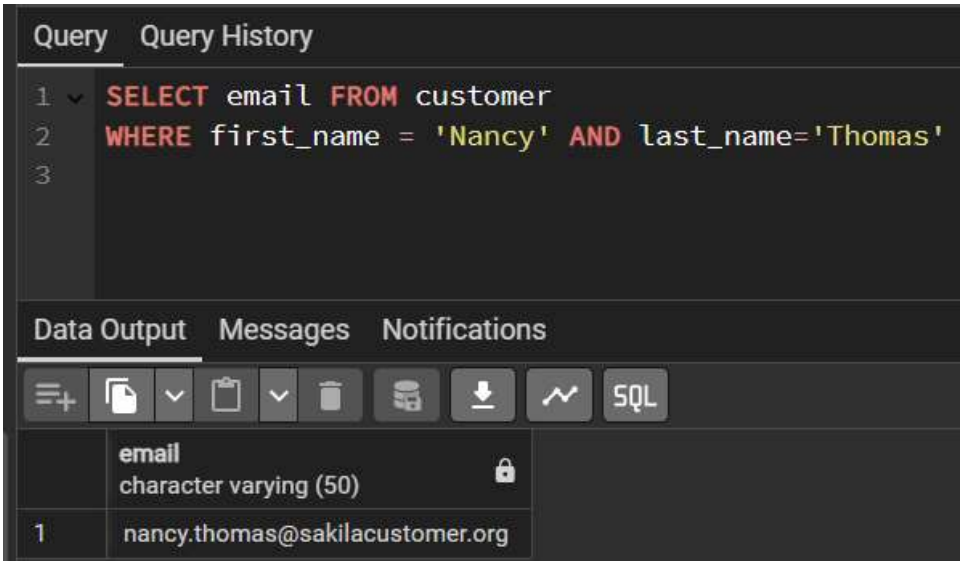
```
SELECT count(*) FROM film
WHERE rental_rate > 4 AND replacement_cost >=19.99 AND rating='R';
```

	count bigint
1	34

⚔️ Challenge "SELECT WHERE"

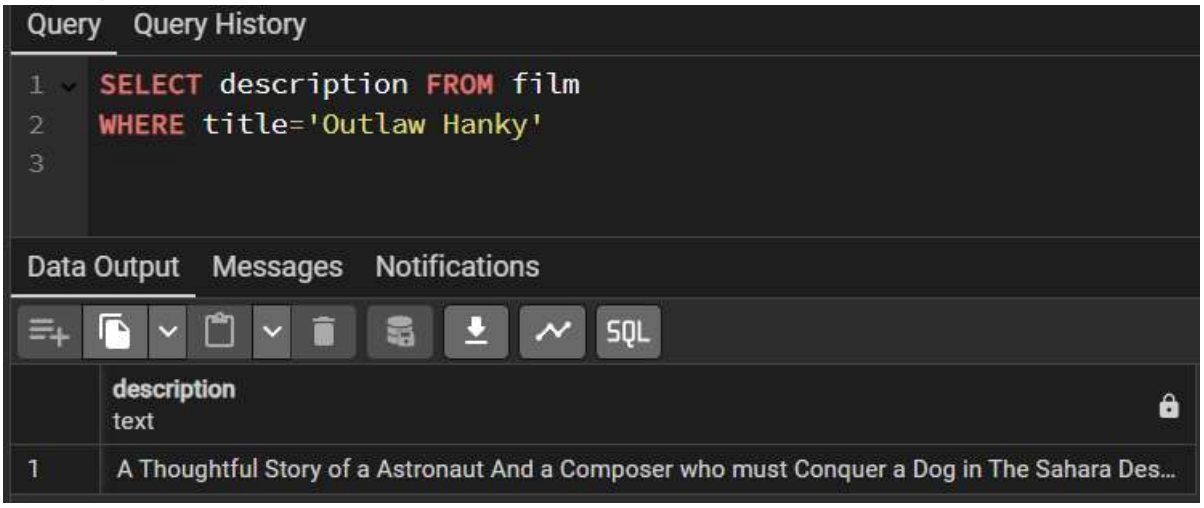
- Ques 1) A customer forgot their wallet at our store! we need to track down their email to inform them. What is the email for the customer with the name Nancy Thomas?

- **Solution)** `SELECT email FROM customer WHERE first_name = 'Nancy' AND last_name='Thomas'`



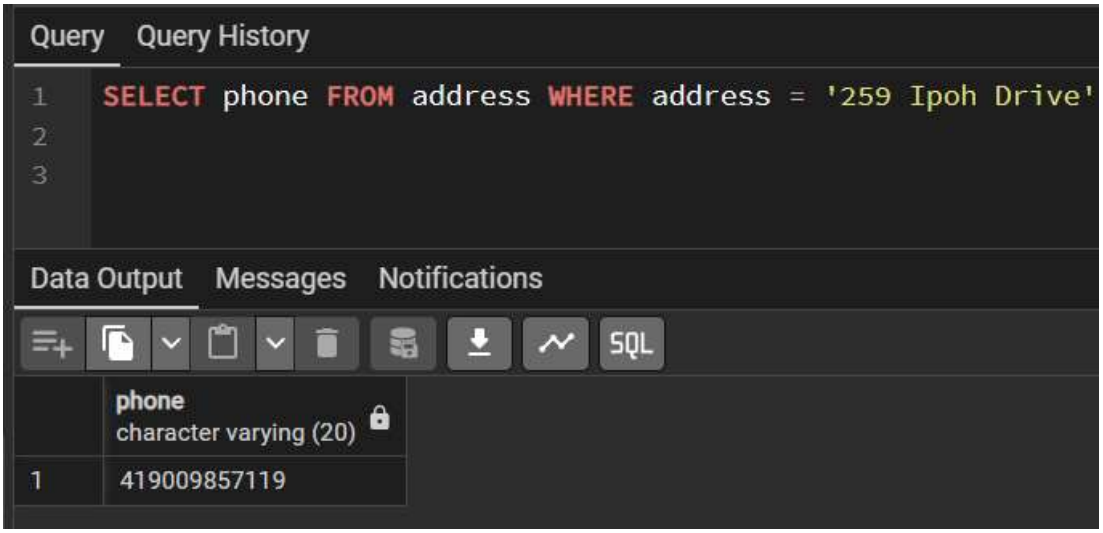
- Ques 2) A customer wants more info about the movie "Outlaw Hanky"

- **Solution)** `SELECT description FROM film WHERE title='Outlaw Hanky'`



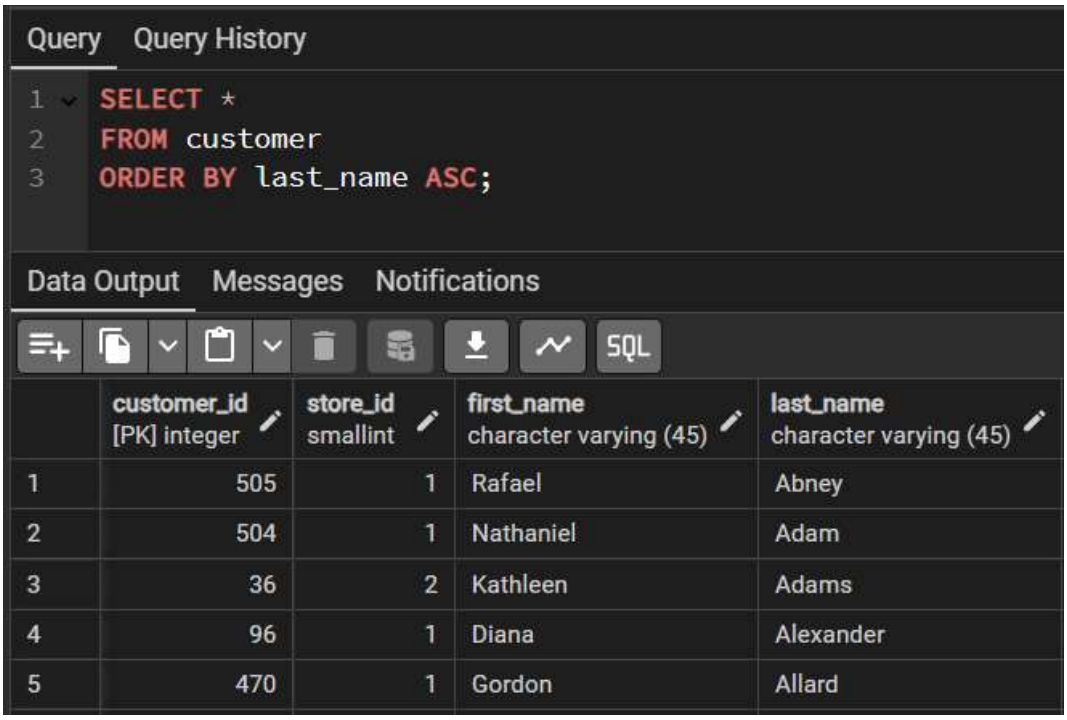
- Ques 3) A customer is late on their movie return, and we've mailed them a letter to their address at '295 Ipoh Drive'. We should call them on phone as well, so retrieve his phone number

- **Solution)** `SELECT phone FROM address WHERE address = '259 Ipoh Drive'`



5. "ORDER BY" command

- The ORDER BY command in SQL is used to sort the result set of a query by one or more columns. You can specify the sort order as ascending (ASC) or descending (DESC).
-



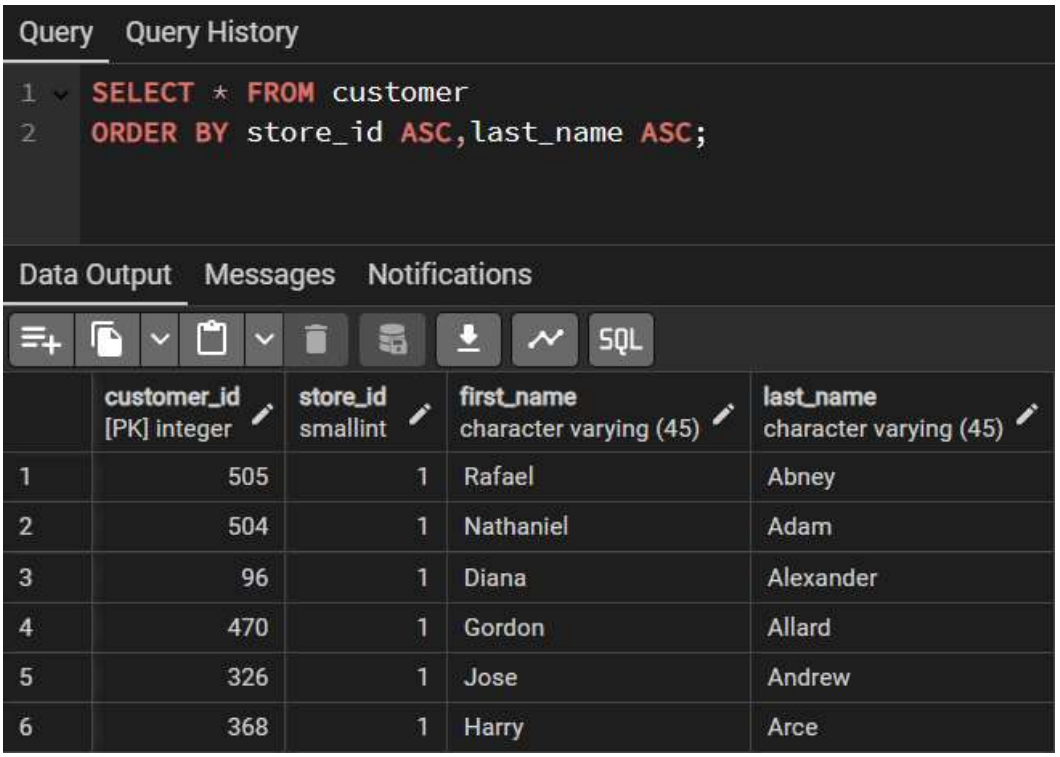
Query | Query History

```
SELECT *
FROM customer
ORDER BY last_name ASC;
```

Data Output | Messages | Notifications

	customer_id [PK] integer	store_id smallint	first_name character varying (45)	last_name character varying (45)
1	505	1	Rafael	Abney
2	504	1	Nathaniel	Adam
3	36	2	Kathleen	Adams
4	96	1	Diana	Alexander
5	470	1	Gordon	Allard

- We can also add more than one column for sorting order
  - e.g. we are here first sorting by store_id and then by last_name
    - `SELECT * FROM customer ORDER BY store_id ASC,last_name ASC;`



Query | Query History

```
SELECT * FROM customer
ORDER BY store_id ASC,last_name ASC;
```

Data Output | Messages | Notifications

	customer_id [PK] integer	store_id smallint	first_name character varying (45)	last_name character varying (45)
1	505	1	Rafael	Abney
2	504	1	Nathaniel	Adam
3	96	1	Diana	Alexander
4	470	1	Gordon	Allard
5	326	1	Jose	Andrew
6	368	1	Harry	Arce

6. "LIMIT" command

- The LIMIT command in SQL is used to specify the maximum number of rows that a query should return. It's commonly used to limit results for performance reasons or to paginate data.
    - `SELECT * FROM payment ORDER BY payment_date DESC LIMIT 5;`

Query | Query History

```
1  SELECT * FROM payment
2  WHERE amount !=0.00
3  ORDER BY payment_date
4  DESC LIMIT 5;
```

Data Output | Messages | Notifications

SQL

	payment_id [PK] integer	customer_id smallint	staff_id smallint	rental_id integer	amount numeric (5,2)	payment_date timestamp without time zone
1	31922	279	2	13538	4.99	2007-05-14 13:44:29.996577
2	31917	267	2	12066	7.98	2007-05-14 13:44:29.996577
3	31919	269	1	13025	3.98	2007-05-14 13:44:29.996577
4	31921	274	1	13486	0.99	2007-05-14 13:44:29.996577
5	31923	282	2	15430	0.99	2007-05-14 13:44:29.996577

⚔️ Challenge "WHERE"

- Ques1) we want to reward our first 10 paying customers, what are the customer ids of those customers?
    - **Solution)** `SELECT customer_id FROM payment  WHERE amount !=0.00 ORDER BY payment_date ASC  LIMIT 10;`

Query Query History

```
SELECT customer_id FROM payment
WHERE amount !=0.00
ORDER BY payment_date ASC
LIMIT 10;
```

Data Output Messages Notifications

	customer_id smallint
1	416
2	516
3	239
4	592
5	49
6	264
7	46
8	481
9	139
10	595

Ques 2) A customer wants to quickly rent a video to watch over their short lunch break. What are the titles of the 5 shortest (runtime) movies?

- **Solution)** `SELECT title,length FROM film ORDER BY length ASC LIMIT 5;`

Query Query History

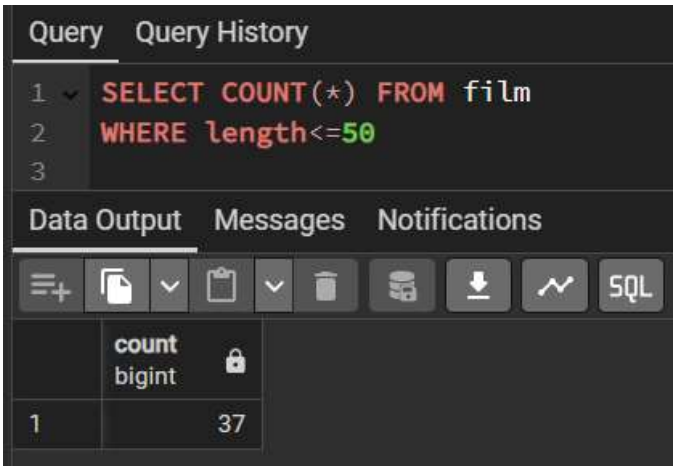
```
SELECT title,length FROM film
ORDER BY length ASC
LIMIT 5;
```

Data Output Messages Notifications

	title character varying (255)	length smallint
1	Labyrinth League	46
2	Alien Center	46
3	Iron Moon	46
4	Kwai Homeward	46
5	Ridgemont Submarine	46

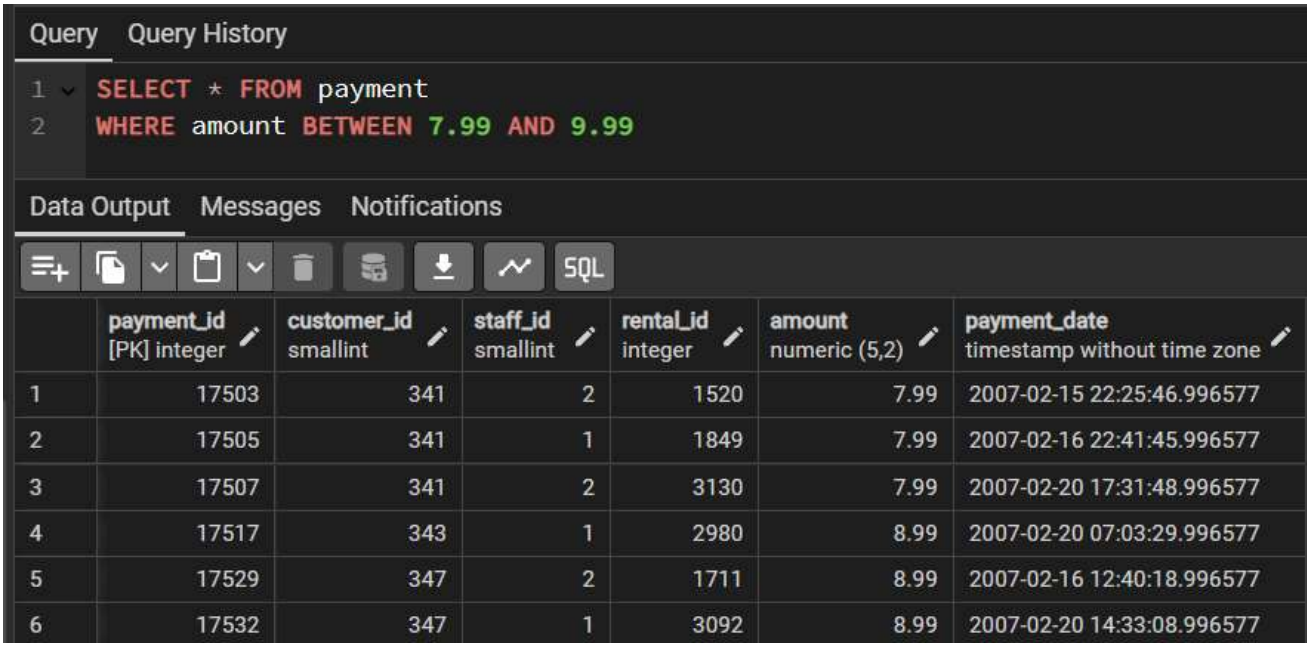
Ques 3) If the previous customer can watch any movie that is 50 minutes or less in run time, how many options does he have?

- **Solution)** `SELECT COUNT(*) FROM film WHERE length<=50`



7. "BETWEEN" command

- Syntax `SELECT column1, column2, ...FROM table_name WHERE column_name BETWEEN value1 AND value2;`
- Both the upper bound and lower bound are included when using the BETWEEN keyword in SQL. This means that the values specified at both ends of the range are part of the result set.



	payment_id [PK] integer	customer_id smallint	staff_id smallint	rental_id integer	amount numeric (5,2)	payment_date timestamp without time zone
1	17503	341	2	1520	7.99	2007-02-15 22:25:46.996577
2	17505	341	1	1849	7.99	2007-02-16 22:41:45.996577
3	17507	341	2	3130	7.99	2007-02-20 17:31:48.996577
4	17517	343	1	2980	8.99	2007-02-20 07:03:29.996577
5	17529	347	2	1711	8.99	2007-02-16 12:40:18.996577
6	17532	347	1	3092	8.99	2007-02-20 14:33:08.996577

- We can also put date but we need to keep a specific format `2024-12-31 or YYYY-MM-DD`
- Note: Date exclude the upper bound

Query Query History

```
SELECT * FROM payment
WHERE payment_date BETWEEN '2007-02-01' AND '2007-02-15'
ORDER BY payment_date DESC
```

Data Output Messages Notifications

SQL

	payment_id [PK] integer	customer_id smallint	staff_id smallint	rental_id integer	amount numeric (5,2)	payment_date timestamp without time zone
1	17743	402	2	1194	4.99	2007-02-14 23:53:34.996577
2	18322	561	2	1193	2.99	2007-02-14 23:52:46.996577
3	19212	186	1	1192	4.99	2007-02-14 23:47:05.996577
4	17617	370	2	1190	6.99	2007-02-14 23:33:58.996577

8. "IN" command

- The IN command in SQL is used to specify multiple values in a WHERE clause. It allows you to filter records based on a list of possible values for a specified column.

Query Query History

```
SELECT * FROM payment
WHERE amount IN (0.99,1.98,1.99)
ORDER BY amount DESC
```

Data Output Messages Notifications

SQL

	payment_id [PK] integer	customer_id smallint	staff_id smallint	rental_id integer	amount numeric (5,2)	payment_date timestamp without time zone
1	17504	341	1	1778	1.99	2007-02-16 17:23:14.996577
2	24123	150	1	12887	1.99	2007-03-19 02:07:20.996577
3	29692	64	1	4690	1.99	2007-04-08 09:32:28.996577
4	21865	514	1	14386	1.99	2007-03-21 08:35:00.996577
5	22874	20	1	15161	1.99	2007-03-22 13:05:48.996577
6	29700	64	2	6698	1.99	2007-04-12 11:13:26.996577
7	18276	546	1	1181	1.99	2007-02-14 23:10:43.996577

- Note: You can also use "NOT IN" to do the iverted operation

Query Query History

```
SELECT * FROM payment
WHERE amount NOT IN (0.99,1.98,1.99)
ORDER BY amount DESC
```

Data Output Messages Notifications

	payment_id [PK] integer	customer_id smallint	staff_id smallint	rental_id integer	amount numeric (5,2)	payment_date timestamp without time zone
1	22650	204	2	15415	11.99	2007-03-22 22:17:22.996577
2	28799	591	2	4383	11.99	2007-04-07 19:14:17.996577
3	20403	362	1	14759	11.99	2007-03-21 21:57:24.996577

Query Query History

```
SELECT * FROM customer
WHERE first_name IN ('Jared','Maria') or last_name IN ('Smith')
```

Data Output Messages Notifications

	customer_id [PK] integer	store_id smallint	first_name character varying (45)	last_name character varying (45)	email character varying (50)
1	524	1	Jared	Ely	jared.ely@sakilacustomer.org
2	1	1	Mary	Smith	mary.smith@sakilacustomer.org
3	7	1	Maria	Miller	maria.miller@sakilacustomer.org

"LIKE" command

The LIKE operator is case-sensitive and is used with wildcards:

- "%" Represents zero or more characters.

Query | Query History

```
SELECT * FROM customer
WHERE first_name LIKE 'Al%'
ORDER BY first_name ASC
```

Data Output | Messages | Notifications

	customer_id [PK] integer	store_id smallint	first_name character varying (45)	last_name character varying (45)	email character varying (50)
1	389	1	Alan	Kahn	alan.kahn@sakilacustomer.org
2	352	1	Albert	Crouse	albert.crouse@sakilacustomer.org
3	568	2	Alberto	Henning	alberto.henning@sakilacustomer.org
4	454	2	Alex	Gresham	alex.gresham@sakilacustomer.org
5	439	2	Alexander	Fennell	alexander.fennell@sakilacustomer....

- "_" Represents a single character

Query | Query History | Scratch

```
SELECT * FROM film
WHERE title LIKE '_er%'
```

Data Output | Messages | Notifications

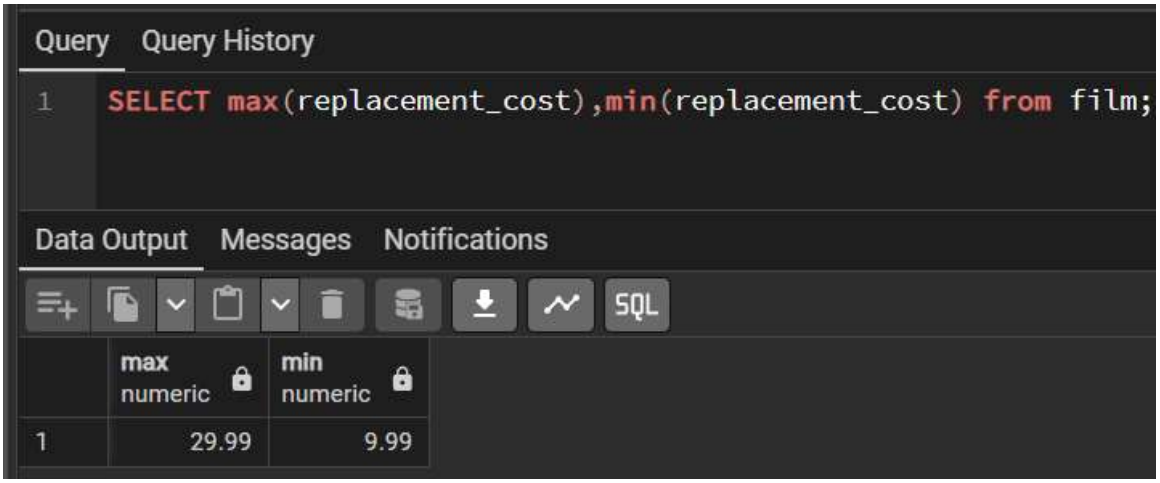
	film_id [PK] integer	title character varying (255)	description text
1	67	Berets Agent	A Taut Saga of a Crocodile And a Boy who must Overcome a Technical Writer in Ancient China
2	308	Ferris Mother	A Touching Display of a Frisbee And a Frisbee who must Kill a Girl in The Gulf of Mexico
3	483	Jericho Mulan	A Amazing Yarn of a Hunter And a Butler who must Defeat a Boy in A Jet Boat

Note: NOT LIKE / NOT ILIKE both are just inverted version of the same

"GROUP BY" command

Aggregate Functions

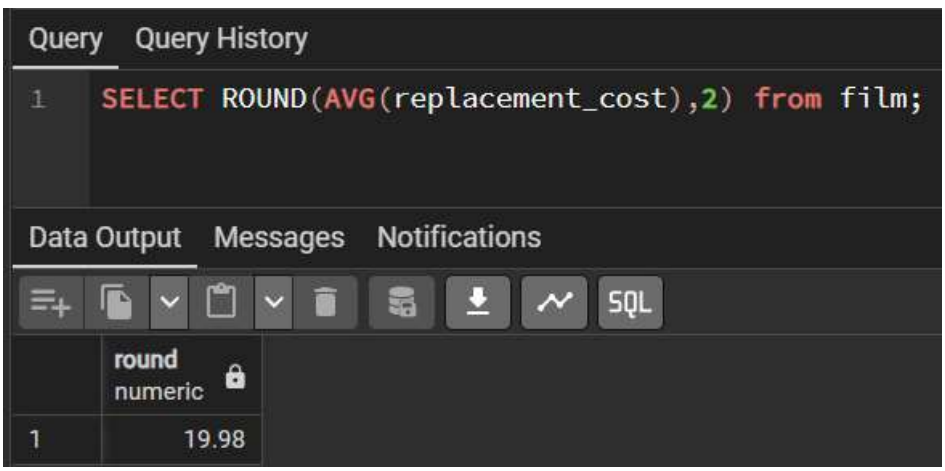
- In SQL, aggregate functions are used to perform calculations on multiple rows of a table and return a single value. Here are the most commonly used aggregate functions:



Function	Description	Example
COUNT()	Counts the number of rows	`SELECT COUNT(*) FROM employees;`
SUM()	Returns the sum of a numeric column	`SELECT SUM(salary) FROM employees;`
AVG()	Returns the average value of a column	`SELECT AVG(salary) FROM employees;`
MAX()	Returns the maximum value	`SELECT MAX(salary) FROM employees;`
MIN()	Returns the minimum value	`SELECT MIN(salary) FROM employees;`
STRING_AGG()	Concatenates values with delimiter	`SELECT STRING_AGG(name, ', ') FROM employees;`

- Some aggregate function may give a long decimal value in return so in order to limit the digits after deciman use "**ROUND**" command

```
ROUND(value,digits after decimal)
```



"GROUP BY"

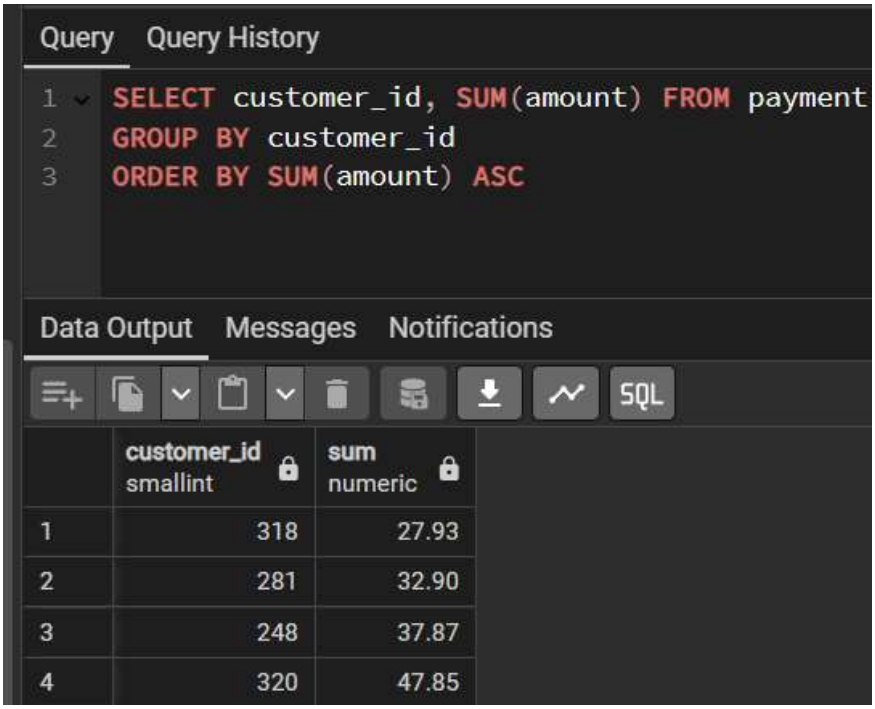
- The GROUP BY statement in SQL is used to group rows that have the same values in specified columns into summary rows, like grouping by a particular field and applying aggregate functions such as COUNT(), SUM(), AVG(), etc. It is often used with aggregate functions to perform calculations on groups of rows.
- Syntax

```
SELECT column1, aggregate_function(column2)
FROM table_name
WHERE condition GROUP BY column1;
```

Examples- GROUP BY

- Ques 1) Find the Amount spent per customer

```
SELECT customer_id, SUM(amount) FROM payment
GROUP BY customer_id
ORDER BY SUM(amount) ASC
```



	customer_id smallint	sum numeric
1	318	27.93
2	281	32.90
3	248	37.87
4	320	47.85

- Ques 2) Find the top 5 customers by numbers of orders they have placed

```
SELECT customer_id, COUNT(amount) FROM payment
GROUP BY customer_id
ORDER BY SUM(amount) DESC
LIMIT 5
```

Query Query History

```
SELECT customer_id, COUNT(amount) FROM payment
GROUP BY customer_id
ORDER BY SUM(amount) DESC
LIMIT 5
```

Data Output Messages Notifications

SQL

	customer_id smallint	count bigint
1	148	45
2	526	42
3	178	39
4	137	38
5	144	40

- Ques 3) Find the amount each customer has spent per staff member of the rental shop

```
SELECT customer_id,staff_id, SUM(amount) FROM payment
GROUP by staff_id,customer_id
ORDEr BY customer_id ASC, staff_id ASC
```

Query Query History

```
SELECT customer_id,staff_id, SUM(amount) FROM payment
GROUP by staff_id,customer_id
ORDEr BY customer_id ASC, staff_id ASC
```

Data Output Messages Notifications

SQL

	customer_id smallint	staff_id smallint	sum numeric
1	1	1	60.85
2	1	2	53.85
3	2	1	55.86
4	2	2	67.88

Ques 4) We want to know the revenue generated each day. So order the dates based on revenue generated

```
SELECT DATE(payment_date),SUM(amount) FROM payment
GROUP BY DATE(payment_date)
ORDER BY SUM(amount) ASC
```

Query Query History

```
SELECT DATE(payment_date),SUM(amount) FROM payment
GROUP BY DATE(payment_date)
ORDER BY SUM(amount) ASC
```

Data Output Messages Notifications

SQL

	date date	sum numeric
1	2007-02-14	116.73
2	2007-04-05	273.36
3	2007-03-16	299.28

⚔️ Group BY challenge

- Ques 1) We have 2 staff members, with staff id 1 and 2. We want to give a bonus to the staff member that handled the most payments [Most in terms of number of payment processed, not total dollar amount]. How many payments did each staff memeber handle and who gets the bonus.

```
SELECT staff_id,COUNT(amount) FROM payment
GROUP BY staff_id
ORDER BY staff_id DESC
```

Query Query History

```
SELECT staff_id,COUNT(amount) FROM payment
GROUP BY staff_id
ORDER BY staff_id DESC
```

Data Output Messages Notifications

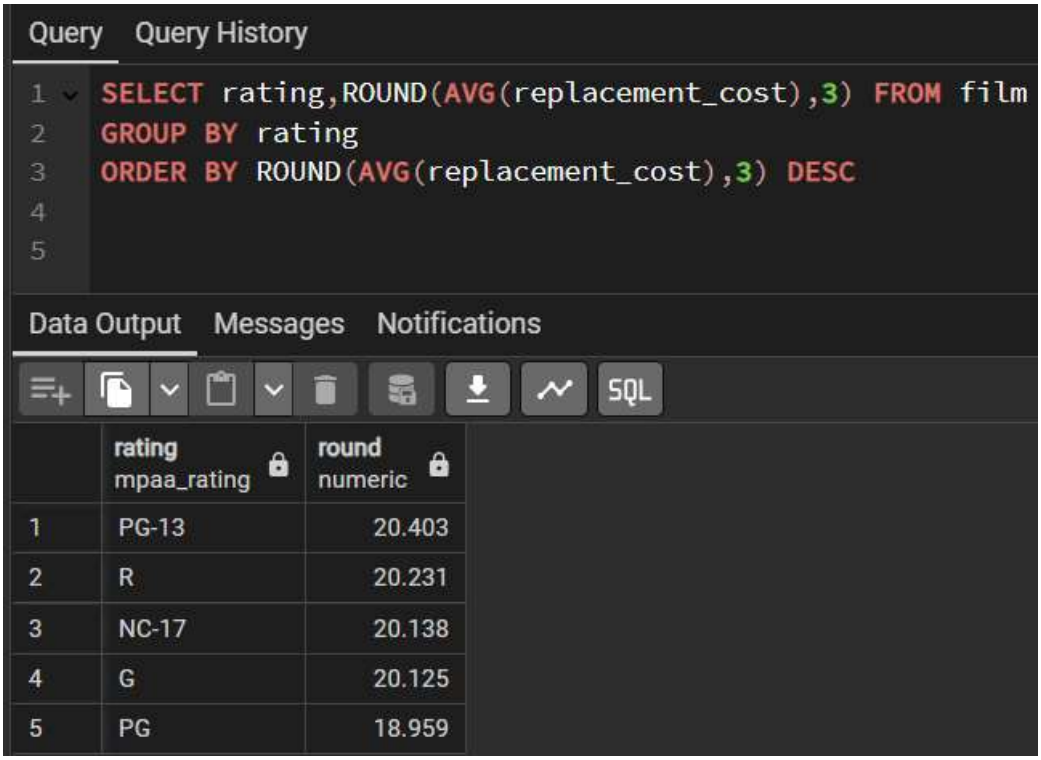
SQL

	staff_id smallint	count bigint
1	2	7304
2	1	7292

- Ques 2) Corporate HQ is connducting a study on the relationship between replacement cost and a movie MPAA rating. what is the average replacement cost per MPAA rating?

```
SELECT rating,ROUND(AVG(replacement_cost),3) FROM film
GROUP BY rating
```

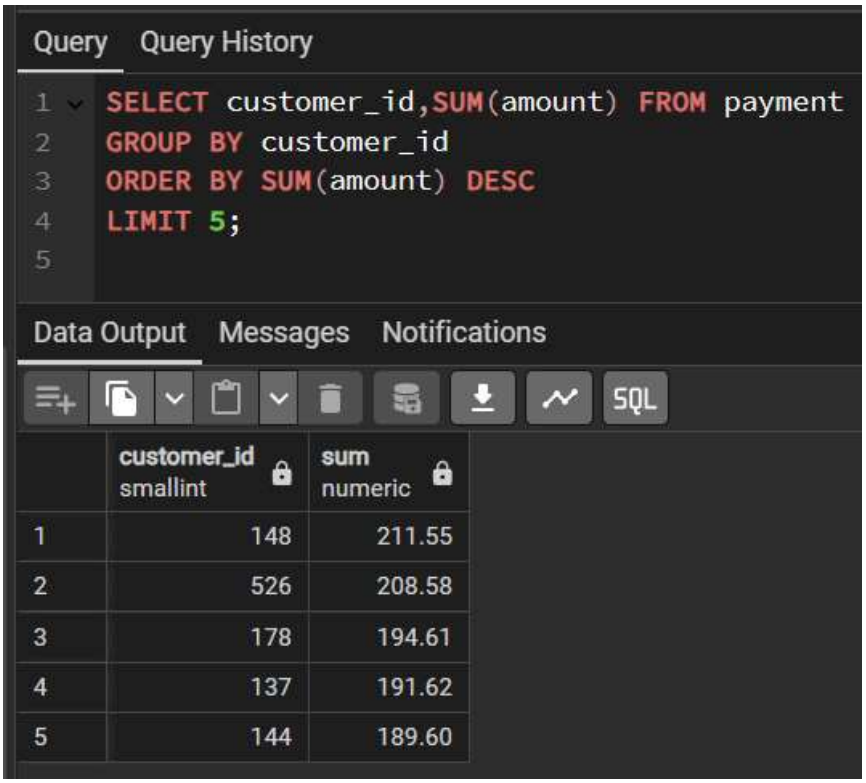
```
ORDER BY ROUND(AVG(replacement_cost),3) DESC
```



	rating mpaa_rating	round numeric
1	PG-13	20.403
2	R	20.231
3	NC-17	20.138
4	G	20.125
5	PG	18.959

- Ques 3) we are running a promotion to reward our top 5 customers with coupons. what are the customer_ids of the top 5 customers by total spend

```
SELECT customer_id,SUM(amount) FROM payment
GROUP BY customer_id
ORDER BY SUM(amount) DESC
LIMIT 5;
```



	customer_id smallint	sum numeric
1	148	211.55
2	526	208.58
3	178	194.61
4	137	191.62
5	144	189.60

"Having" clause

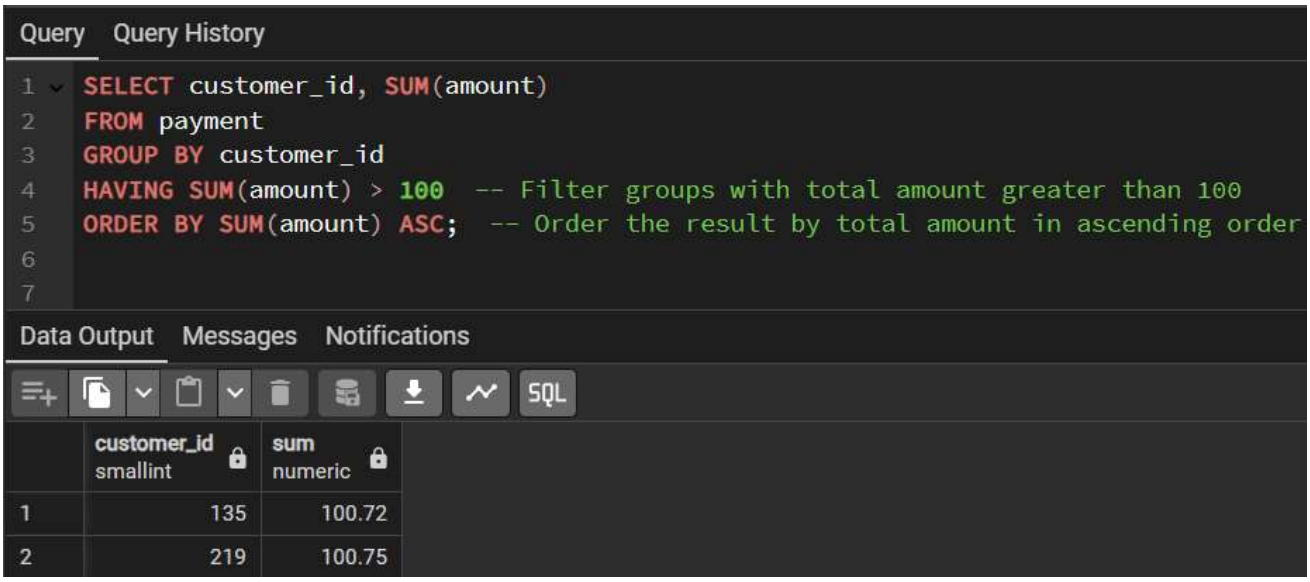
- The HAVING clause in SQL is used to filter the results of a GROUP BY query based on aggregate functions. It's similar to the WHERE clause, but HAVING is used after grouping, while WHERE is used before grouping.

```
SELECT column1, aggregate_function(column2)
FROM table_name
GROUP BY column1
HAVING condition;
```

Key Differences Between WHERE and HAVING:

- WHERE: Filters rows before the grouping.
    - Therefore we cannot use the WHERE clause after an **aggregate function** because WHERE filters rows before any grouping or aggregation takes place.
- HAVING: Filters groups after the grouping.

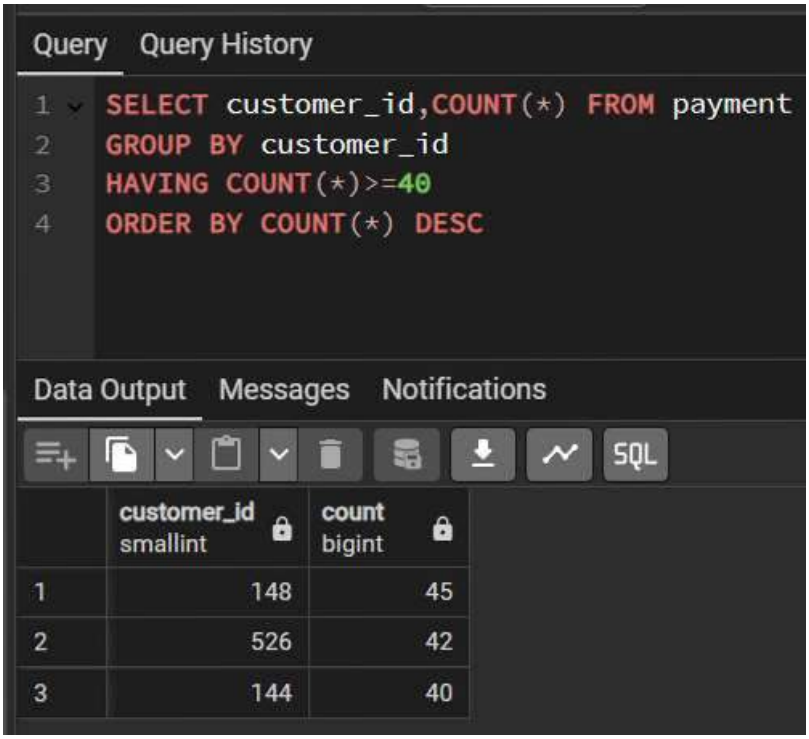
Note: you cannot use the HAVING clause after the ORDER BY clause in the same query



⚔️ HAVING challenge

- Ques 1) we are launching a platinum service for our most loyal customers. We will assign platinum status to customers that have had 40 or more transaction payments. What customer_ids are eligible for platinum status

```
SELECT customer_id,COUNT(*) FROM payment
GROUP BY customer_id
HAVING COUNT(*)>=40
ORDER BY COUNT(*) DESC
```



- Ques 2) what are the customer ids of customers who have spent more than $100 in payment transactions with our staff_id member 2?

Method: 1

```
SELECT customer_id,staff_id,SUM(amount) FROM payment
GROUP BY customer_id,staff_id
HAVING SUM(amount)>100 AND staff_id=2
ORDER BY SUM(amount) DESC;
```

Query | Query History

```
SELECT customer_id,staff_id,SUM(amount) FROM payment
GROUP BY customer_id,staff_id
HAVING SUM(amount)>100 AND staff_id=2
ORDER BY SUM(amount) DESC;
```

Data Output | Messages | Notifications

	customer_id smallint	staff_id smallint	sum numeric
1	187	2	110.81
2	522	2	102.80
3	526	2	101.78
4	211	2	108.77
5	148	2	110.78

Method: 2

```
SELECT customer_id,SUM(amount) FROM payment
WHERE staff_id =2
GROUP BY Customer_id
HAVING SUM(amount)>100
ORDER BY SUM(amount) DESC
```

Query | Query History

```
1  SELECT customer_id,SUM(amount) FROM payment
2  WHERE staff_id =2
3  GROUP BY Customer_id
4  HAVING SUM(amount)>100
5  ORDER BY SUM(amount) DESC
6
```

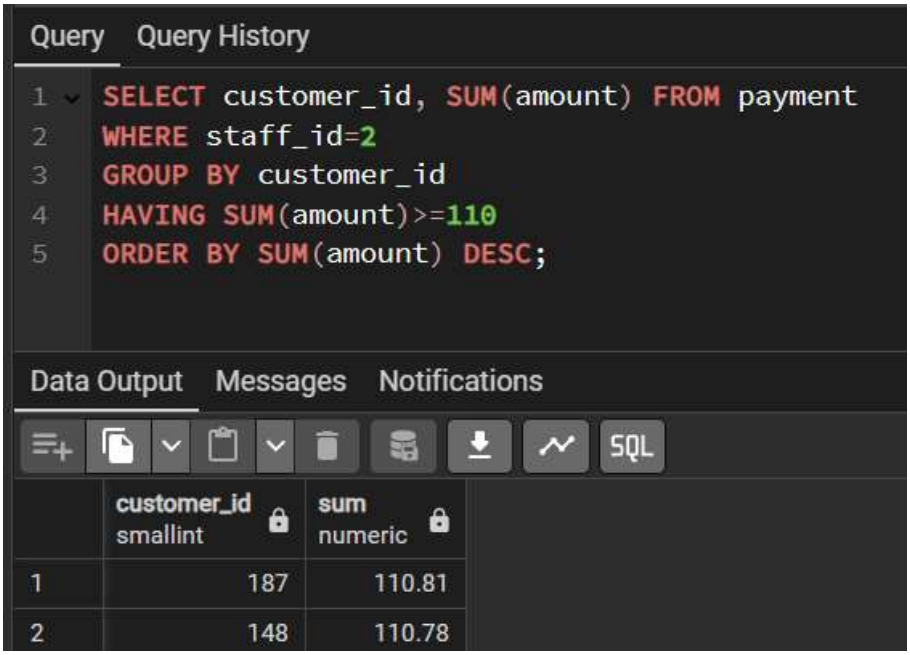
Data Output | Messages | Notifications

	customer_id smallint	sum numeric
1	187	110.81
2	148	110.78
3	211	108.77
4	522	102.80
5	526	101.78

Assesment TEST-1

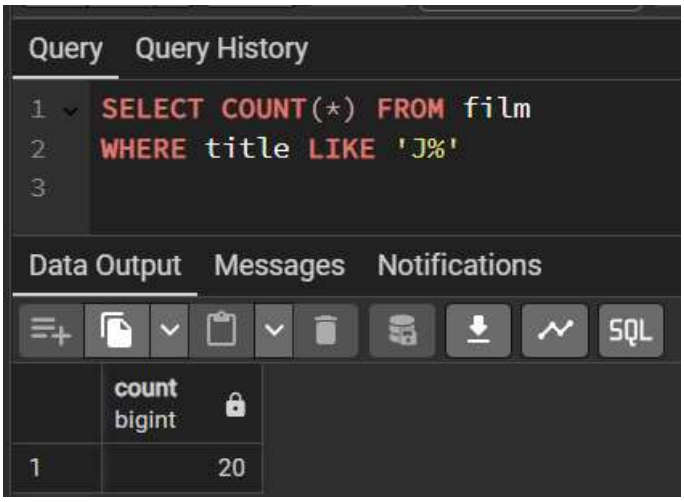
1. Return the customer IDs of customers who have spent at least $110 with the staff member who has an ID of 2.

```
SELECT customer_id, SUM(amount) FROM payment
WHERE staff_id=2
GROUP BY customer_id
HAVING SUM(amount)>=110
ORDER BY SUM(amount) DESC;
```



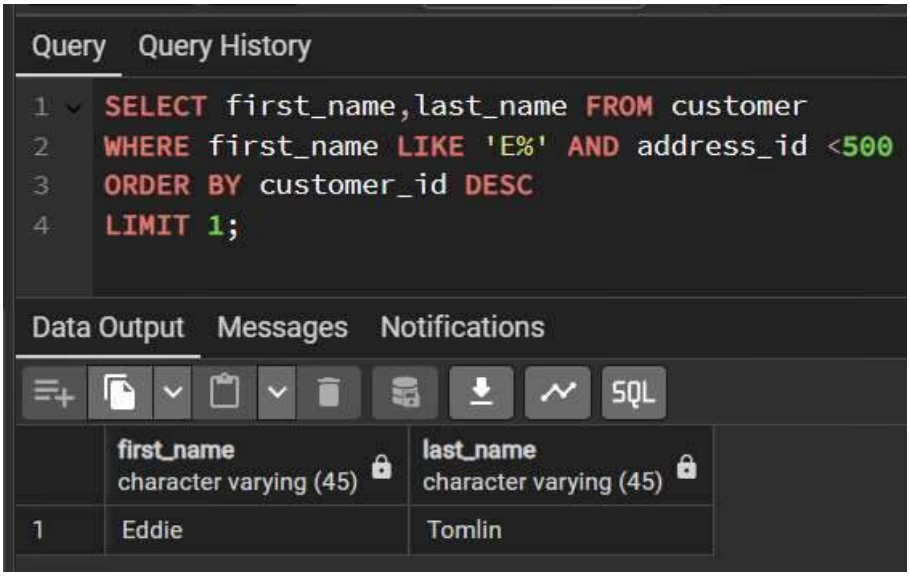
2. How many films begin with the letter J?

```
SELECT COUNT(*) FROM film
WHERE title LIKE 'J%'
```



3. What customer has the highest customer ID number whose name starts with an 'E' and has an address ID lower than 500?

```sql
SELECT first_name,last_name FROM customer
WHERE first_name LIKE 'E%' AND address_id <500
ORDER BY customer_id DESC
LIMIT 1;
```

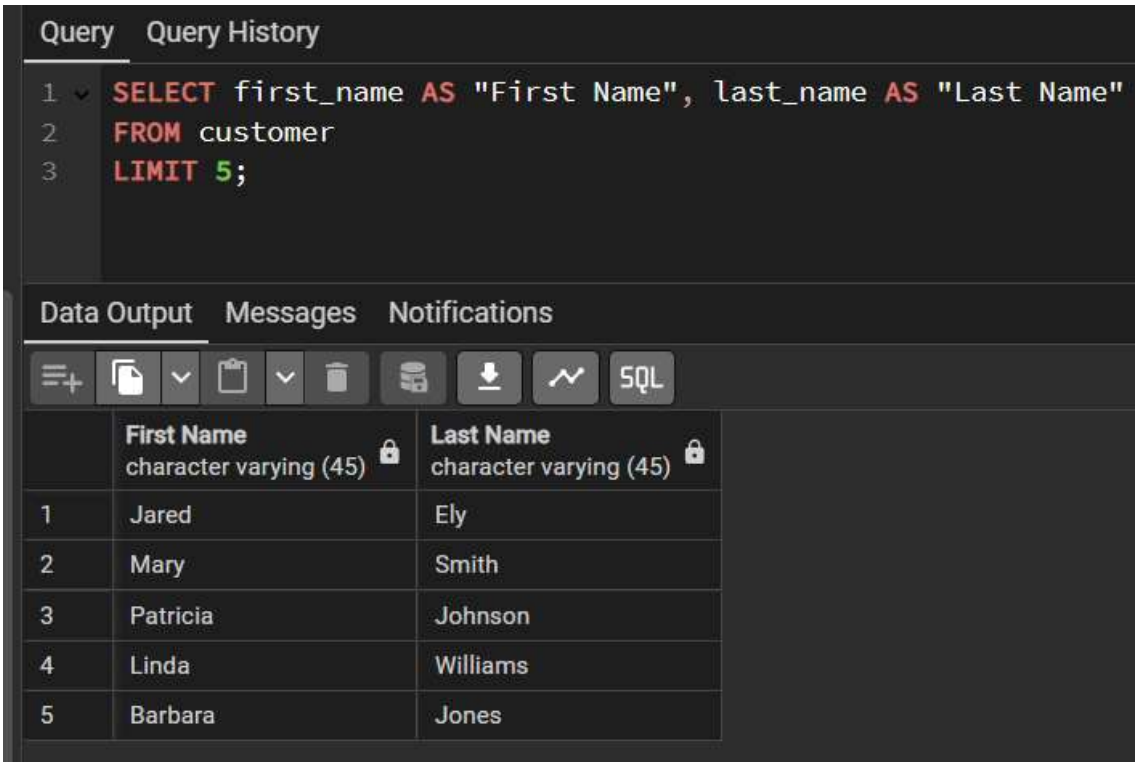


"JOIN" command

"AS" Clause

- In SQL, the AS keyword is used to alias (give a temporary name to) columns or tables in a query. This can make the output more readable, especially when dealing with complex expressions or joins. The alias exists only for the duration of the query.

```
SELECT column_name AS alias_name
FROM table_name;
```

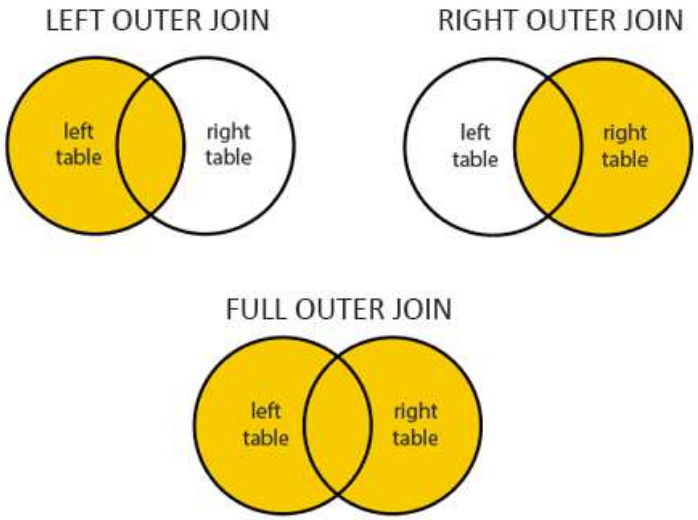


- **Note: AS operator gets executed at the very end of a query, meaning that we can not use the ALIAS inside WHERE operator**

JOIN

- In SQL, the JOIN command is used to combine rows from two or more tables based on a related column between them. There are different types of joins, such as INNER JOIN, LEFT JOIN, RIGHT JOIN, and FULL JOIN, each serving different purposes.

Types of Joins:

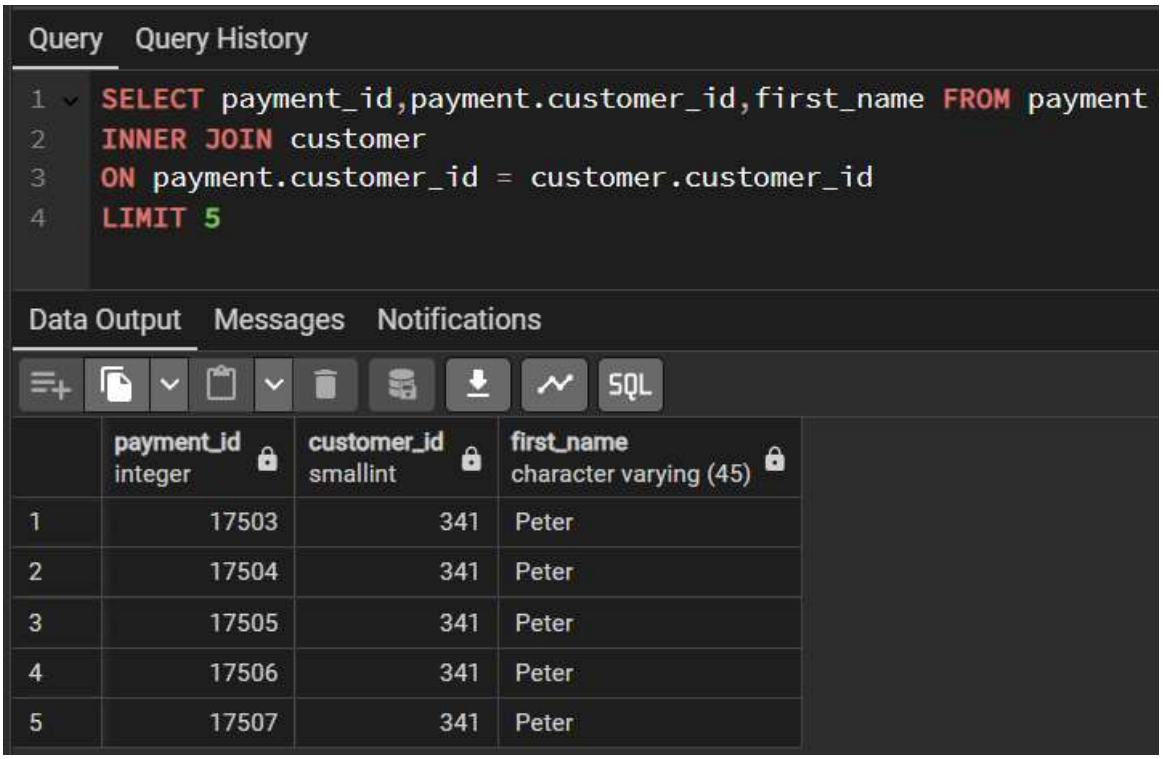


1. INNER JOIN [Intersection]

- Returns only the rows where there is a match in both tables.

```
SELECT table1.column1, table2.column2
FROM table1
INNER JOIN table2
ON table1.common_column = table2.common_column;
```

Note: If the column is unique to only one table only then we dont need to mention tha table name but if both table has a particular column common then we have to define the table from where we want that column to be pulled



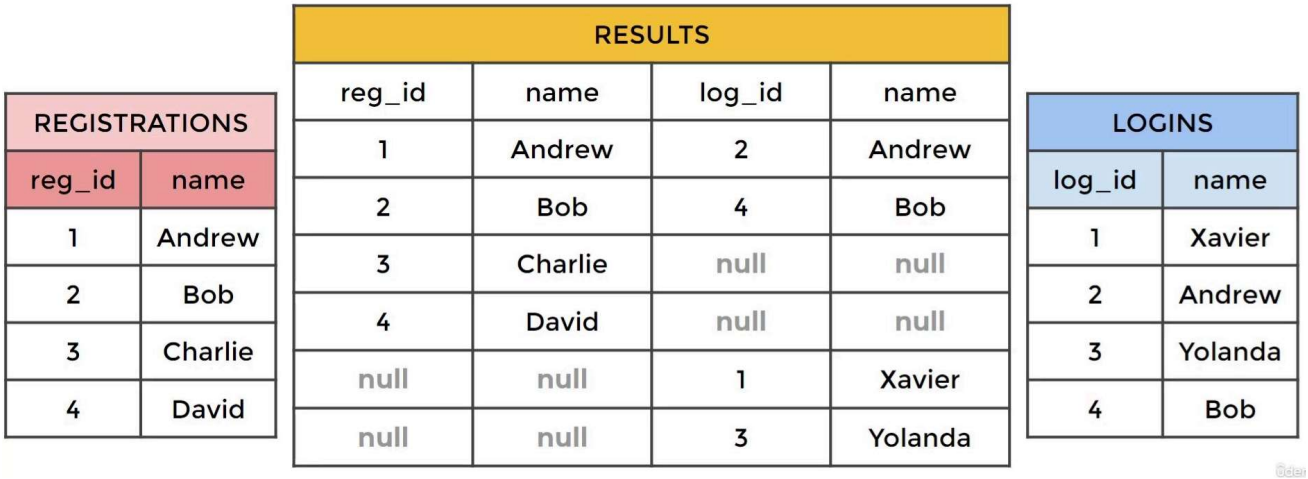
	payment_id integer	customer_id smallint	first_name character varying (45)
1	17503	341	Peter
2	17504	341	Peter
3	17505	341	Peter
4	17506	341	Peter
5	17507	341	Peter

2. FULL OUTER JOIN

- FULL OUTER JOIN (also called a FULL JOIN) in SQL returns all rows when there is a match in either the left or right table. If there is no match, it will still return rows from both tables, with NULL values in place for the missing matches.

```
SELECT column1, column2, ...
FROM table1
FULL OUTER JOIN table2
ON table1.common_column = table2.common_column;
```

- SELECT * FROM Registrations FULL OUTER JOIN Logins
  ON Registrations.name = Logins.name



REGISTRATIONS

reg_id	name
1	Andrew
2	Bob
3	Charlie
4	David

RESULTS

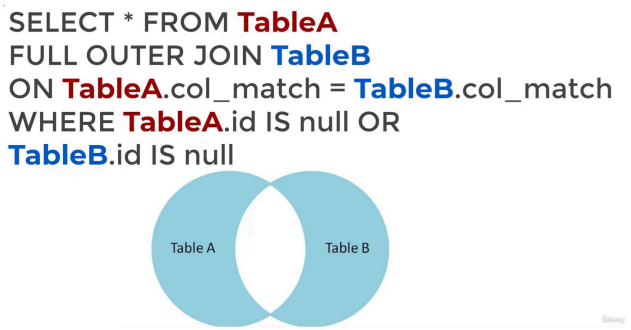
reg_id	name	log_id	name
1	Andrew	2	Andrew
2	Bob	4	Bob
3	Charlie	null	null
4	David	null	null
null	null	1	Xavier
null	null	3	Yolanda

LOGINS

log_id	name
1	Xavier
2	Andrew
3	Yolanda
4	Bob

- FULL OUTER JOIN + WHERE NULL = Opposite(INNER JOIN) = UNIQUE Rows

```
SELECT column1, column2, ...
FROM table1
FULL OUTER JOIN table2
ON table1.common_column = table2.common_column;
WHERE table1.common_column IS null OR table2.common_column IS null
```



- SELECT * FROM Registrations FULL OUTER JOIN Logins
  ON Registrations.name = Logins.name
  WHERE Registrations.reg_id IS null OR
  Logins.log_id IS null

REGISTRATIONS

reg_id	name
1	Andrew
2	Bob
3	Charlie
4	David

RESULTS

reg_id	name	log_id	name
3	Charlie	null	null
4	David	null	null
null	null	1	Xavier
null	null	3	Yolanda

LOGINS

log_id	name
1	Xavier
2	Andrew
3	Yolanda
4	Bob

3. LEFT JOIN (or LEFT OUTER JOIN)

- Returns all the rows from the left table and the matching rows from the right table. If there's no match, NULL is returned for the right table's columns.

```
SELECT table1.column1, table2.column2
FROM table1         -- Table 1 will be the left table [since it is mentioned first]
```

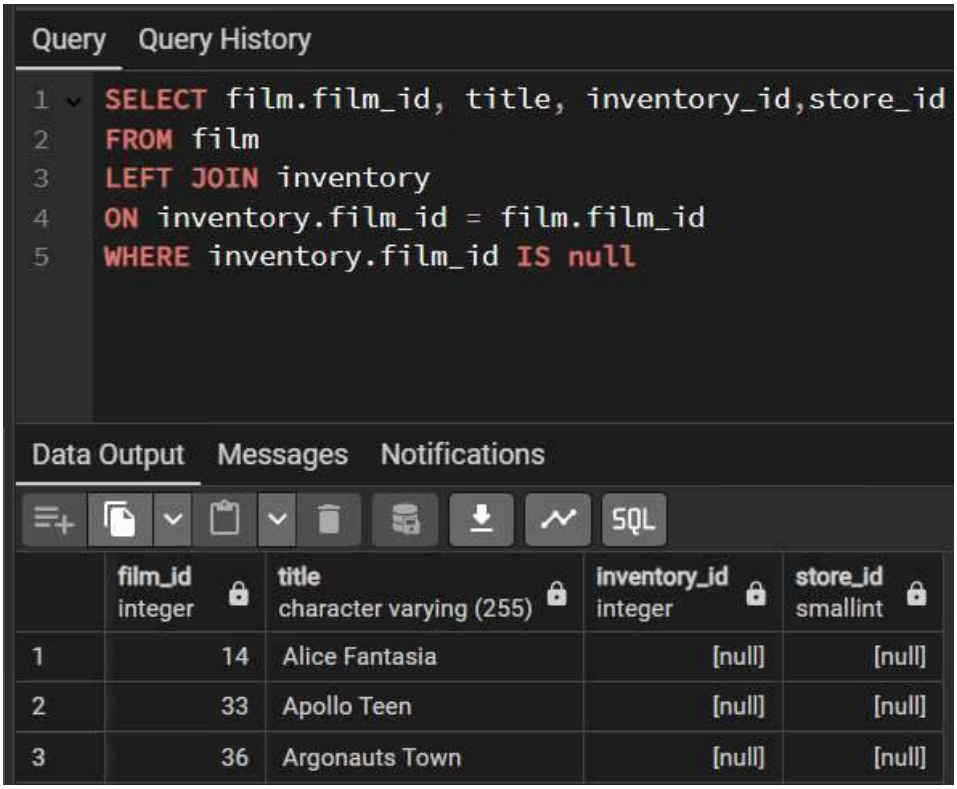
```
LEFT JOIN table2    -- or you can use "LEFT OUTER JOIN"
ON table1.common_column = table2.common_column;
```

- LEFT OUTER JOIN + WHERE not found in B = Exclusive Table 1

```
SELECT * FROM Table1
LEFT OUTER JOIN Table2
ON Table1.col_match = Table2.col_match
WHERE Table2.id is null
```

- Practice Question
    - What SQL query would you use to retrieve a list of films from the film table that do not have any associated records in the inventory table, ensuring that films without any inventory are identified?

```
SELECT film.film_id, title, inventory_id,store_id
FROM film
LEFT JOIN inventory
ON inventory.film_id = film.film_id
WHERE inventory.film_id IS null
```



	film_id integer	title character varying (255)	inventory_id integer	store_id smallint
1	14	Alice Fantasia	[null]	[null]
2	33	Apollo Teen	[null]	[null]
3	36	Argonauts Town	[null]	[null]

4. RIGHT JOIN (or RIGHT OUTER JOIN)

- Returns all the rows from the right table and the matching rows from the left table. If there's no match, NULL is returned for the left table's columns.

```
SELECT table1.column1, table2.column2
FROM table1
RIGHT JOIN table2
ON table1.common_column = table2.common_column;
```

5. FULL JOIN (or FULL OUTER JOIN)

- Returns all rows when there is a match in either table. If there's no match, NULL values are returned from the non-matching table.

```
SELECT table1.column1, table2.column2
FROM table1
FULL JOIN table2
ON table1.common_column = table2.common_column;
```

"UNION" Command

- The UNION command in SQL is used to combine the results of two or more SELECT queries into a single result set. By default, the UNION operator removes duplicate rows from the final result. If you want to include duplicates, you can use UNION ALL.

```
SELECT column1, column2, ...
FROM table1
UNION
SELECT column1, column2, ...
FROM table2;
```

⚔️ JOIN challenge

- Ques 1) mail every customers who live in california about an upcoming sales there

```
SELECT first_name,last_name,district,email FROM customer
INNER JOIN address
ON address.address_id = customer.address_id
WHERE address.district = 'California'
```

Query Query History

```
SELECT first_name,last_name,district,email FROM customer
INNER JOIN address
ON address.address_id = customer.address_id
WHERE address.district = 'California'
```

Data Output Messages Notifications

	first_name character varying (45)	last_name character varying (45)	district character varying (20)	email character varying (50)
1	Patricia	Johnson	California	patricia.johnson@sakilacustomer.org
2	Betty	White	California	betty.white@sakilacustomer.org
3	Alice	Stewart	California	alice.stewart@sakilacustomer.org
4	Rosa	Reynolds	California	rosa.reynolds@sakilacustomer.org
5	Renee	Lane	California	renee.lane@sakilacustomer.org
6	Kristin	Johnston	California	kristin.johnston@sakilacustomer.org
7	Cassandra	Walters	California	cassandra.walters@sakilacustomer.org
8	Jacob	Lance	California	jacob.lance@sakilacustomer.org
9	Rene	Mcalister	California	rene.mcalister@sakilacustomer.org

Ques 2) A customer walks in and is a huge fan of the actor "Nick Wahlberg" and wants to know which movie he is in. Get a list of all of his movies.

```
SELECT actor.actor_id,first_name,Last_name,film_actor.film_id,title from actor
INNER JOIN film_actor
ON actor.actor_id = film_actor.actor_id
INNER JOIN film
ON film_actor.film_id = film.film_id
WHERE first_name='Nick' AND last_name ='Wahlberg'
```

- film [title name, film_id] == actor [actor_id, Name of actor] == film_actor[film_id,actor_id
  - we need 2 join
  - [1st one to join film_actor and actor based on actor_id which will give us "actor_name" and "film_id"]

- After than we can use film_id to merge film database to get title names based on film_id

Query Query History

```
SELECT actor.actor_id,first_name,Last_name,film_actor.film_id,title from a
INNER JOIN film_actor
ON actor.actor_id = film_actor.actor_id
INNER JOIN film
ON film_actor.film_id = film.film_id
WHERE first_name='Nick' AND last_name ='Wahlberg'
```

Data Output Messages Notifications

	actor_id integer	first_name character varying (45)	last_name character varying (45)	film_id smallint	title character varying (255)
1	2	Nick	Wahlberg	3	Adaptation Holes
2	2	Nick	Wahlberg	31	Apache Divine
3	2	Nick	Wahlberg	47	Baby Hall
4	2	Nick	Wahlberg	105	Bull Shawshank
5	2	Nick	Wahlberg	132	Chainsaw Uptown
6	2	Nick	Wahlberg	145	Chisum Behavior
7	2	Nick	Wahlberg	226	Destiny Saturday
8	2	Nick	Wahlberg	249	Dracula Crystal
9	2	Nick	Wahlberg	314	Fight Jawbreaker
10	2	Nick	Wahlberg	321	Flash Wars
11	2	Nick	Wahlberg	357	Gilbert Pelican
12	2	Nick	Wahlberg	369	Goodfellas Salute
13	2	Nick	Wahlberg	399	Happiness United

Advance SQL

Timestamps and Extract functionality

In SQL, there are various data types and functions for working with **time** and **date** values. These help in storing, retrieving, and manipulating temporal data like timestamps, dates, or times. Here's an overview of how you can work with them:

Date and Time Data Types:

1. **DATE**: Stores date values (`YYYY-MM-DD` format).
    - Example: `'2024-09-24'`
2. **TIME**: Stores time values (`HH:MM:SS` format).
    - Example: `'08:45:00'`
3. **DATETIME / TIMESTAMP**: Stores both date and time values (`YYYY-MM-DD HH:MM:SS` format).
    - Example: `'2024-09-24 08:45:00'`
4. **INTERVAL**: Used to represent a span of time (e.g., days, hours).

Common Date and Time Functions:

1. `CURRENT_DATE`

- Returns the current date.

```sql
SELECT CURRENT_DATE;
```

2. `CURRENT_TIME`

- Returns the current time.

```sql
SELECT CURRENT_TIME;
```

3. `CURRENT_TIMESTAMP` or `NOW()`

- Returns the current date and time.

```sql
SELECT CURRENT_TIMESTAMP;
-- or
SELECT NOW();
```

4. Extract Parts of Date or Time

- You can extract specific parts of a date or time (e.g., year, month, day).

```sql
SELECT EXTRACT(YEAR FROM CURRENT_DATE);   -- Extracts year

SELECT EXTRACT(QUARTER FROM CURRENT_DATE);-- Extract Quarter
 - SELECT EXTRACT(QUARTER FROM '2024-09-24') AS quarter_number;

SELECT EXTRACT(MONTH FROM CURRENT_DATE);  -- Extracts month

SELECT EXTRACT(WEEK FROM CURRENT_DATE);   -- Extarct week
 - - SELECT EXTRACT(WEEK FROM '2024-09-24') AS week_number;

SELECT EXTRACT(DAY FROM CURRENT_DATE);    -- Extracts day

SELECT EXTRACT(HOUR FROM CURRENT_TIME);   -- Extracts hour
```

5. DATE_ADD / ADDDATE:

- Add an interval to a date.

```sql
SELECT DATE_ADD('2024-09-24', INTERVAL 7 DAY);  -- Adds 7 days
```

6. DATE_SUB / SUBDATE:

- Subtract an interval from a date.

```sql
SELECT DATE_SUB('2024-09-24', INTERVAL 1 MONTH);  -- Subtracts 1 month
```

7. DATEDIFF:

- Calculate the difference between two dates.

```sql
SELECT DATEDIFF('2024-09-24', '2024-09-01');  -- Returns number of days between two dates
```

8. FORMAT Date/Time

- You can format dates and times using `DATE_FORMAT()` in MySQL.

```sql
SELECT DATE_FORMAT(NOW(), '%Y-%m-%d %H:%i:%s');  -- Formats the date and time
```

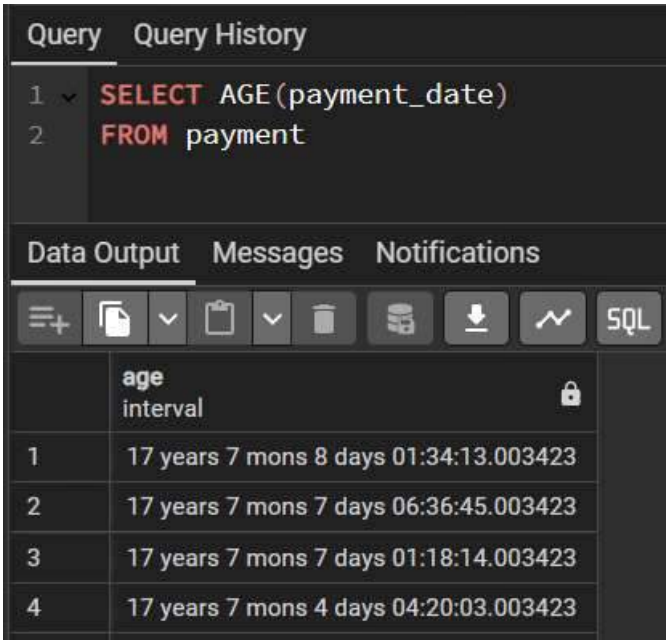
9. SHOW TIMEZONE

- To show the current time zone you are on

```sql
SHOW TIMEZONE -- return Asia/Calcutta
```

10. Age

```sql
SELECT AGE(payment_date)
FROM payment
```



Example Queries:

1. Get the current date:

```
SELECT CURRENT_DATE;
```

2. Find all records from the past 30 days:

```
SELECT *
FROM orders
WHERE order_date >= DATE_SUB(CURRENT_DATE, INTERVAL 30 DAY);
```

3. Find the difference in days between two dates:

```
SELECT DATEDIFF('2024-10-01', '2024-09-24') AS days_difference;
```

4. Retrieve records where the time is after a certain point in the day:

```
SELECT *
FROM schedule
WHERE appointment_time > '15:00:00';
```

TO CHAR

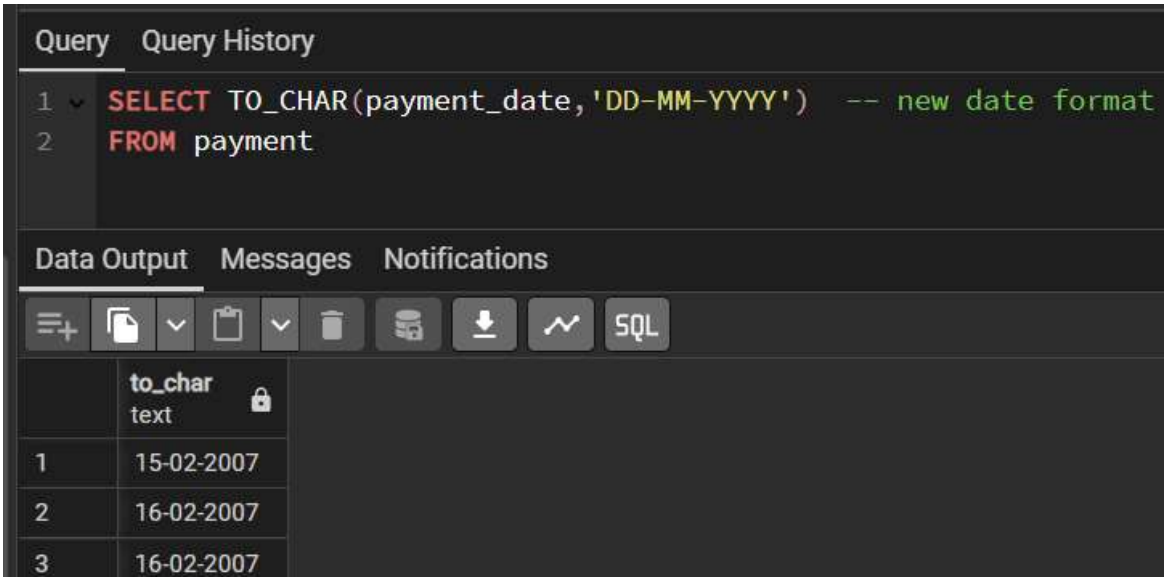
- The TO_CHAR function in SQL is used to convert date, time, and numeric values into a string format. This is especially useful when you want to format dates or numbers in a specific way for display or reporting purposes.

```
TO_CHAR(expression, format)
```

```
SELECT TO_CHAR(SYSDATE, 'YYYY-MM-DD') AS formatted_date
FROM dual;
```

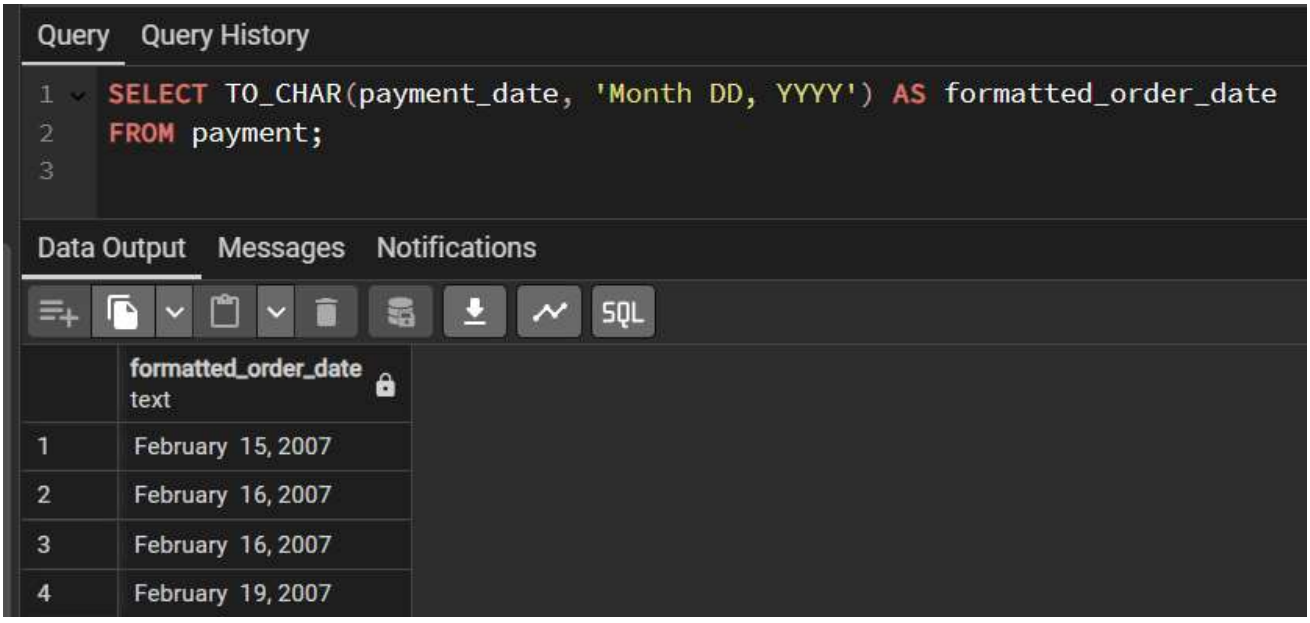
- Example 1

```
SELECT TO_CHAR(payment_date,'DD-MM-YYYY')  -- new date format
FROM payment
```



- Example 2

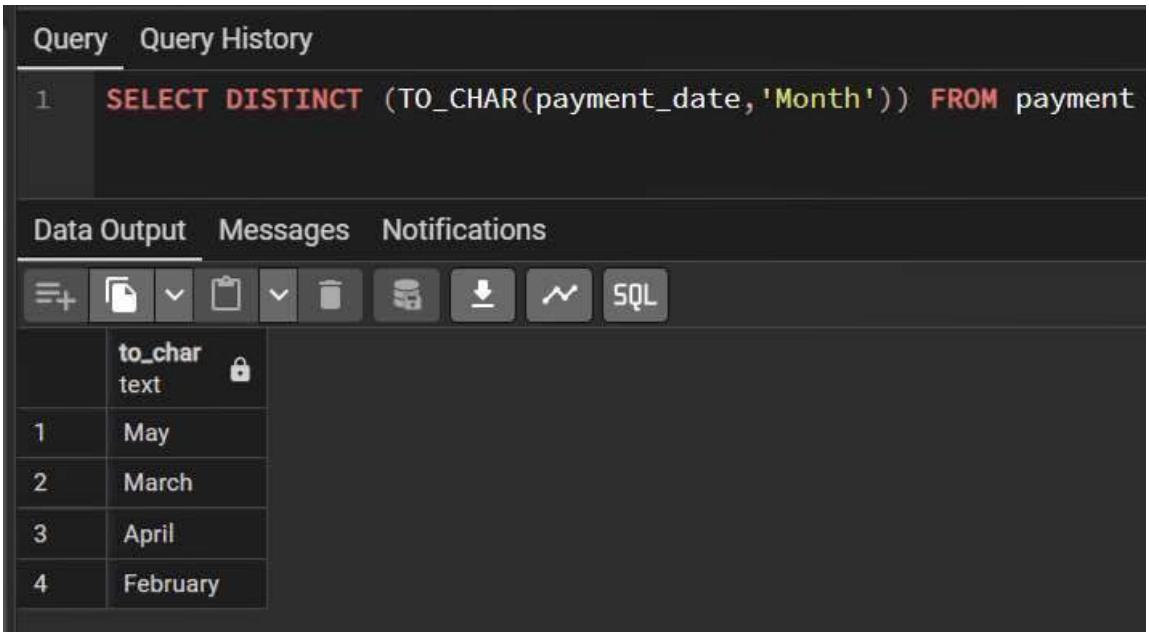
```
SELECT TO_CHAR(payment_date, 'Month DD, YYYY') AS formatted_order_date
FROM payment;
```



⚔️ Timestamp and Extract challenge

Ques 1) During which months did payments occurs and format the answer to return back the full month name

```
SELECT DISTINCT (TO_CHAR(payment_date,'Month')) FROM payment
```



Ques 2) How many payment did occur on monday

Solution: 1

```
SELECT COUNT(*) FROM payment
WHERE EXTRACT(dow FROM payment_date)=1
```

- dow = day of the week 0 = Sunday 1 = Monday 2 = Tuesday and so on

Solution: 2

```
SELECT COUNT(*)
FROM payment
WHERE TO_CHAR(payment_date, 'D') = '2';
```

1 = Sunday 2 = Monday 3 = Tuesday 4 = Wednesday 5 = Thursday 6 = Friday

Mathematical function

```
SELECT title,rental_rate,
replacement_cost,
ROUND(rental_rate*100/replacement_cost,2) AS Rental_percentage
FROM film
ORDER BY ROUND(rental_rate*100/replacement_cost,2) DESC
```

Query | Query History

```sql
SELECT title,rental_rate,
replacement_cost,
ROUND(rental_rate*100/replacement_cost,2) AS Rental_percentage
FROM film
ORDER BY ROUND(rental_rate*100/replacement_cost,2) DESC
```

Data Output | Messages | Notifications

SQL

	title character varying (255)	rental_rate numeric (4,2)	replacement_cost numeric (5,2)	rental_percentage numeric
1	Truman Crazy	4.99	9.99	49.95
2	Daisy Menagerie	4.99	9.99	49.95
3	North Tequila	4.99	9.99	49.95
4	Sting Personal	4.99	9.99	49.95

String Functions

Common SQL String Functions

1. **`LENGTH / LEN`**
   - **Description**: Returns the length of a string.
   - **Syntax**:
     - PostgreSQL/Oracle: `LENGTH(string)`
     - SQL Server: `LEN(string)`
   - **Example**:

```sql
SELECT LENGTH('Hello') AS length;  -- Returns 5 in PostgreSQL/Oracle
SELECT LEN('Hello') AS length;     -- Returns 5 in SQL Server
```

2. **`UPPER / LOWER`**
   - **Description**: Converts a string to upper or lower case.
   - **Syntax**:
     - `UPPER(string)`
     - `LOWER(string)`
   - **Example**:

```sql
SELECT UPPER('hello') AS upper_case;  -- Returns 'HELLO'
SELECT LOWER('HELLO') AS lower_case;  -- Returns 'hello'
```

3. **`SUBSTRING / SUBSTR`**
   - **Description**: Extracts a substring from a string.

- **Syntax**:
  - PostgreSQL: `SUBSTRING(string FROM start FOR length)`
  - MySQL: `SUBSTR(string, start, length)`
  - SQL Server: `SUBSTRING(string, start, length)`
- **Example**:

```
SELECT SUBSTRING('Hello World', 1, 5) AS substring;  -- Returns 'Hello'
SELECT SUBSTR('Hello World', 7, 5) AS substring;     -- Returns 'World' (MySQL)
```

4. **`TRIM`**
   - **Description**: Removes leading and trailing spaces from a string.
   - **Syntax**:
     - `TRIM(string)`
   - **Example**:

```
SELECT TRIM('   Hello World   ') AS trimmed;  -- Returns 'Hello World'
```

5. **`REPLACE`**
   - **Description**: Replaces all occurrences of a substring within a string with another substring.
   - **Syntax**:
     - `REPLACE(string, old_substring, new_substring)`
   - **Example**:

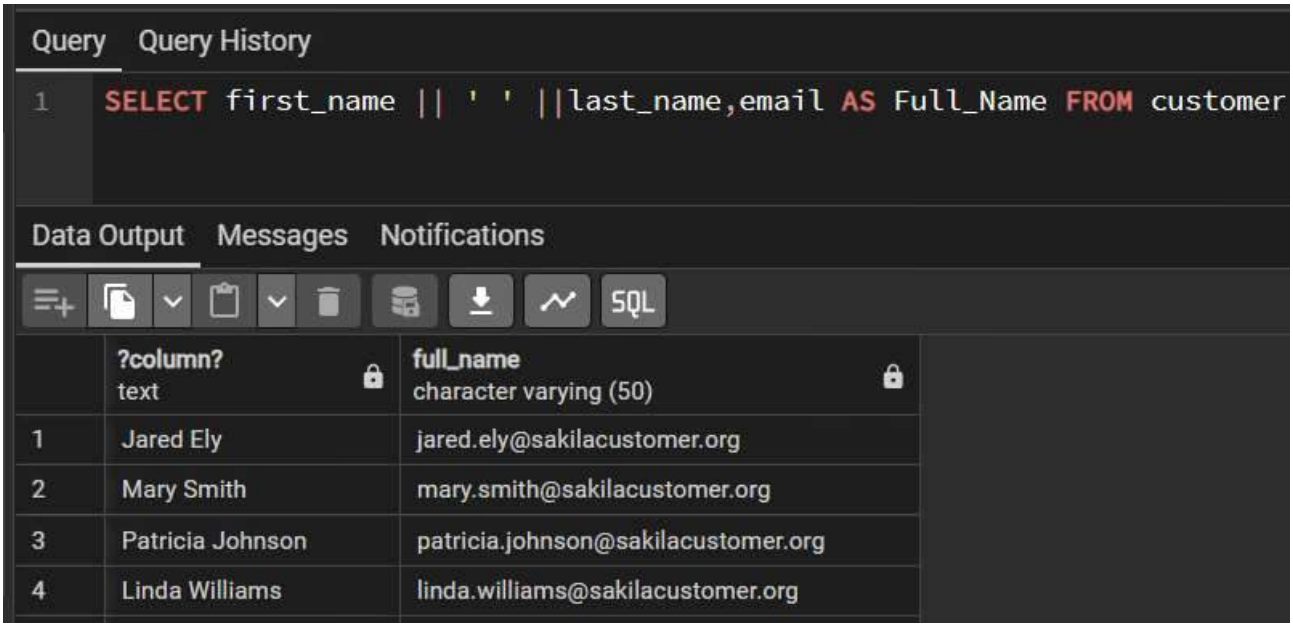
```
SELECT REPLACE('Hello World', 'World', 'SQL') AS replaced;  -- Returns 'Hello SQL'
```

6. **`CONCAT`**
   - **Description**: Concatenates two or more strings together.
   - **Syntax**:
     - `CONCAT(string1, string2, ...)`
   - **Example**:

```
SELECT CONCAT('Hello', ' ', 'World') AS concatenated;  -- Returns 'Hello World'
```

- **Example**:

```
SELECT first_name || ' ' ||last_name,email AS Full_Name FROM customer
```



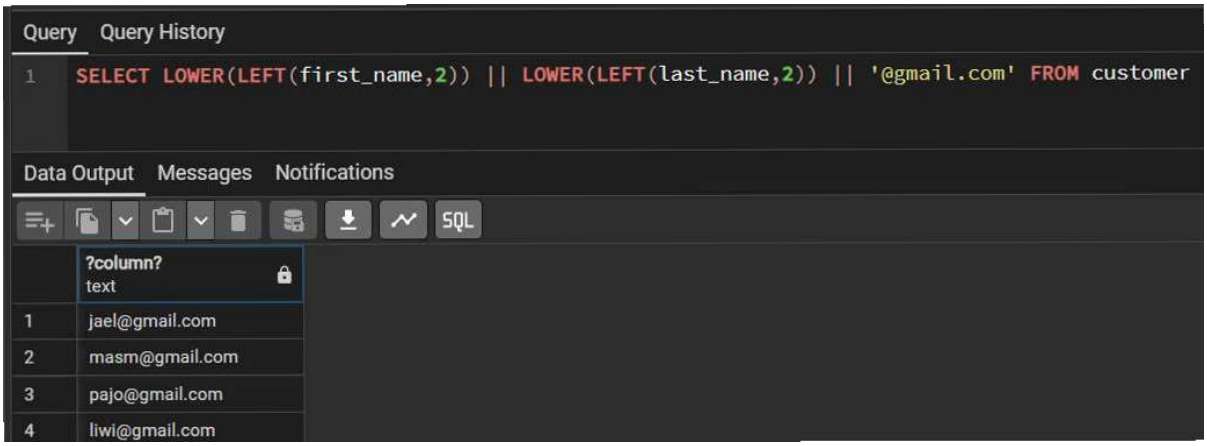

```
SELECT CONCAT(first_name, ' ',last_name),email AS Full_Name FROM customer
```

7. **`CHARINDEX / INSTR`**
   - **Description**: Returns the position of a substring within a string.
   - **Syntax**:
     - SQL Server: `CHARINDEX(substring, string)`
     - MySQL: `INSTR(string, substring)`
   - **Example**:

```
SELECT CHARINDEX('World', 'Hello World') AS position;  -- Returns 7 (SQL Server)
SELECT INSTR('Hello World', 'World') AS position;      -- Returns 7 (MySQL)
```

8. **`LEFT / RIGHT`**
   - **Description**: Returns a specified number of characters from the left or right side of a string.
   - **Syntax**:
     - SQL Server: `LEFT(string, length)` and `RIGHT(string, length)`
   - **Example**:

```
SELECT LEFT('Hello', 2) AS left_part;  -- Returns 'He'
SELECT RIGHT('Hello', 2) AS right_part;  -- Returns 'lo'
SELECT LOWER(LEFT(first_name,2)) || LOWER(LEFT(last_name,2)) || '@gmail.com' FROM customer
```



subQuery

- A subquery, also known as a nested query or inner query, is a query within another SQL query. Subqueries can be used in various places, such as in the SELECT, FROM, WHERE, or HAVING clauses.

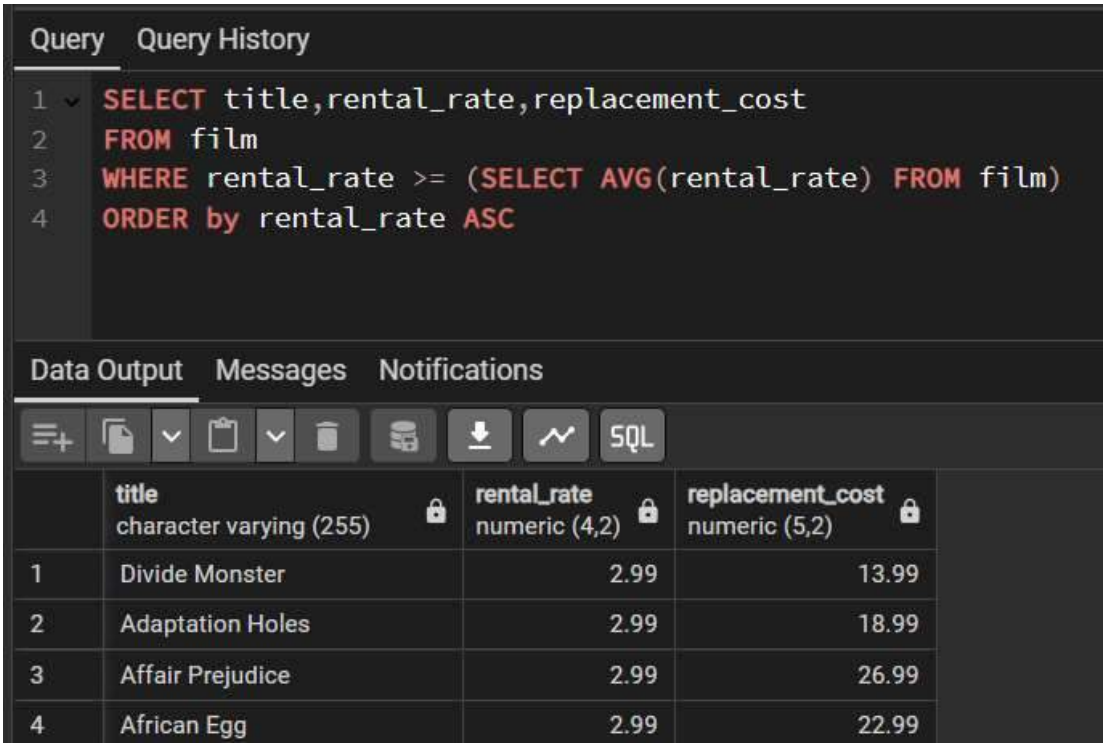
Note: Subquery is executed first

Example -1

```
SELECT customer_id,amount
FROM payment
WHERE amount = (SELECT MAX(amount) FROM payment);
```

Example -2

```
SELECT title,rental_rate,replacement_cost
FROM film
WHERE rental_rate >= (SELECT AVG(rental_rate) FROM film)
ORDER by rental_rate ASC
```



	title character varying (255)	rental_rate numeric (4,2)	replacement_cost numeric (5,2)
1	Divide Monster	2.99	13.99
2	Adaptation Holes	2.99	18.99
3	Affair Prejudice	2.99	26.99
4	African Egg	2.99	22.99

EXIST

- The EXISTS operator in SQL is used to test whether a subquery returns any rows. It returns TRUE if the subquery produces any result, and FALSE if it doesn't return any rows. EXISTS is often used in combination with subqueries, and it is a way to check for the existence of rows that meet a certain

```
SELECT customer_id, first_name, last_name
FROM customers
WHERE EXISTS (
    SELECT *
    FROM orders
    WHERE customers.customer_id = orders.customer_id);
```

SELF JOIN

- A self join is a type of join where a table is joined with itself. This can be useful when you want to compare rows within the same table, such as comparing employees with their managers or finding pairs of related data points.

Note: When using a self join it is necessary to use an alias for the table, otherwise the table names would be ambigious.

```
SELECT f1.title,f2.title,f1.length FROM film as f1
INNER JOIN film AS f2 ON
f1.film_id != f2.film_id AND f1.length = f2.length
```

Query Query History

```
1  SELECT f1.title,f2.title,f1.length FROM film as f1
2  INNER JOIN film AS f2 ON
3  f1.film_id != f2.film_id AND f1.length = f2.length
4
5
```

Data Output Messages Notifications

	title character varying (255)	title character varying (255)	length smallint
1	Chamber Italian	Resurrection Silverado	117
2	Chamber Italian	Magic Mallrats	117
3	Chamber Italian	Graffiti Love	117

Assessment -2

Download the Database for this assignment

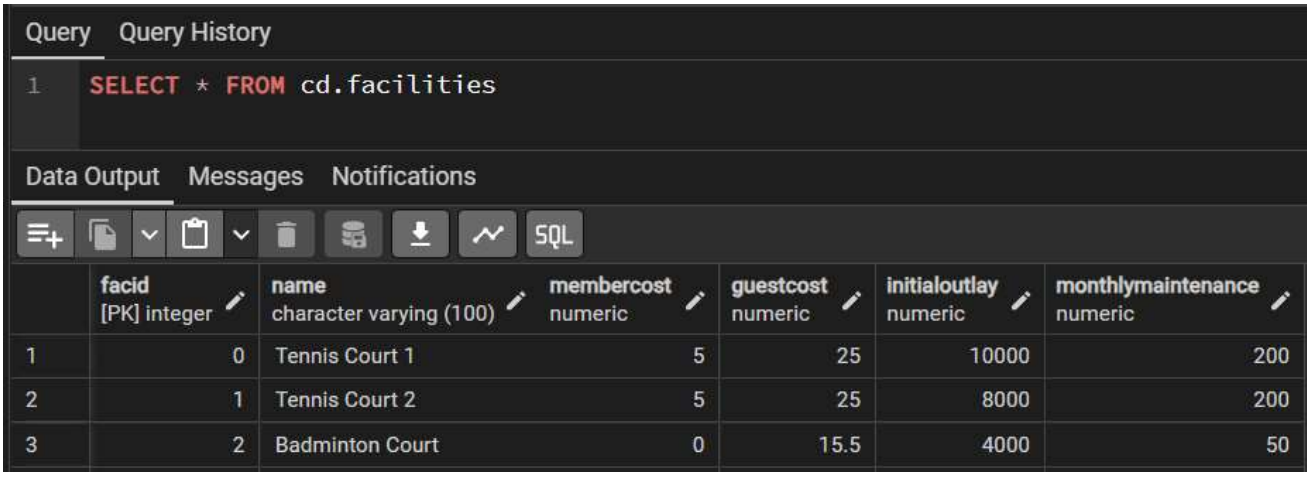
```
https://drive.google.com/file/d/1wDqIK6zt5twWnCOx97ywipaiWR2d0OfT/view
```

Create a new database and restore the database with this above tar file [For more detailed step please refer the Setup Section of the repo]

Ques 1) How can you retrieve all the information from the cd.facilities table?

- Since the database has 2 schemas[cd, public] we need to define the schema name along with the table name to retrieve it

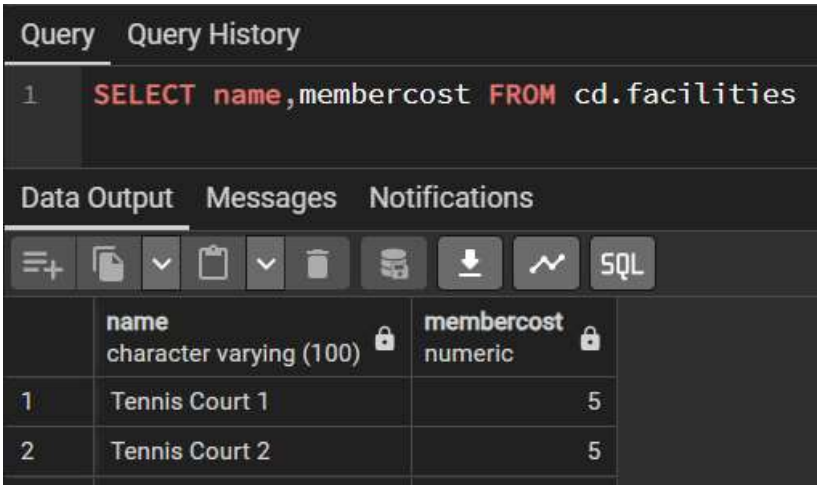
```
SELECT * FROM cd.facilities
```



	facid [PK] integer	name character varying (100)	membercost numeric	guestcost numeric	initialoutlay numeric	monthlymaintenance numeric
1	0	Tennis Court 1	5	25	10000	200
2	1	Tennis Court 2	5	25	8000	200
3	2	Badminton Court	0	15.5	4000	50

Ques 2) You want to print out a list of all of the facilities and their cost to members. How would you retrieve a list of only facility names and costs?

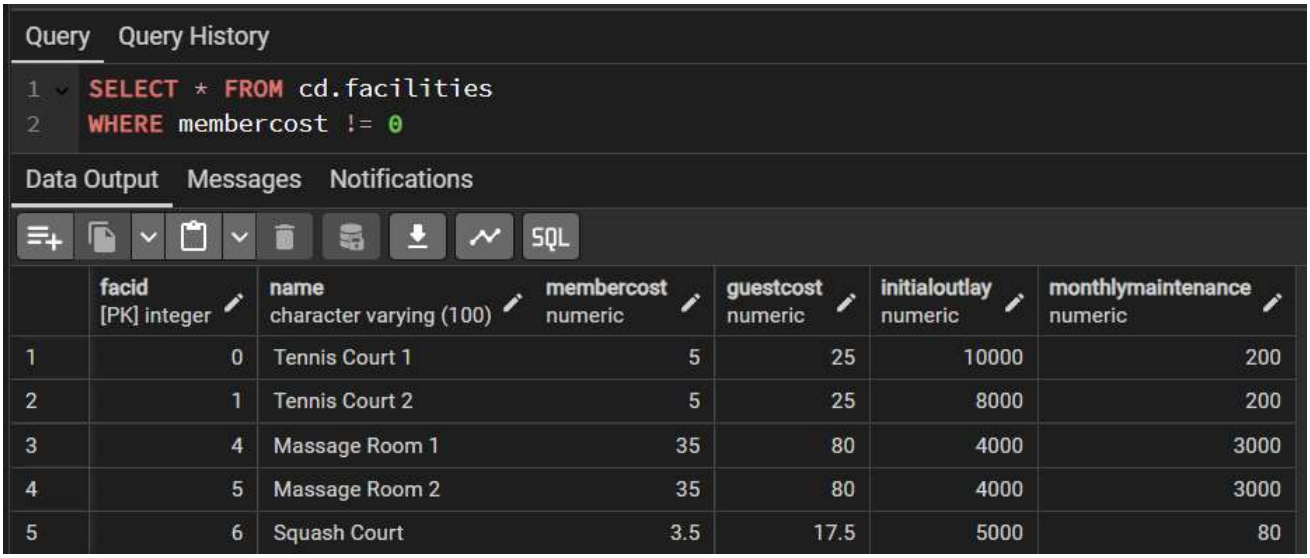
```
SELECT name,membercost FROM cd.facilities
```



	name character varying (100)	membercost numeric
1	Tennis Court 1	5
2	Tennis Court 2	5

Ques 3) How can you produce a list of facilities that charge a fee to members?

```
SELECT * FROM cd.facilities
WHERE membercost != 0
```



Query | Query History

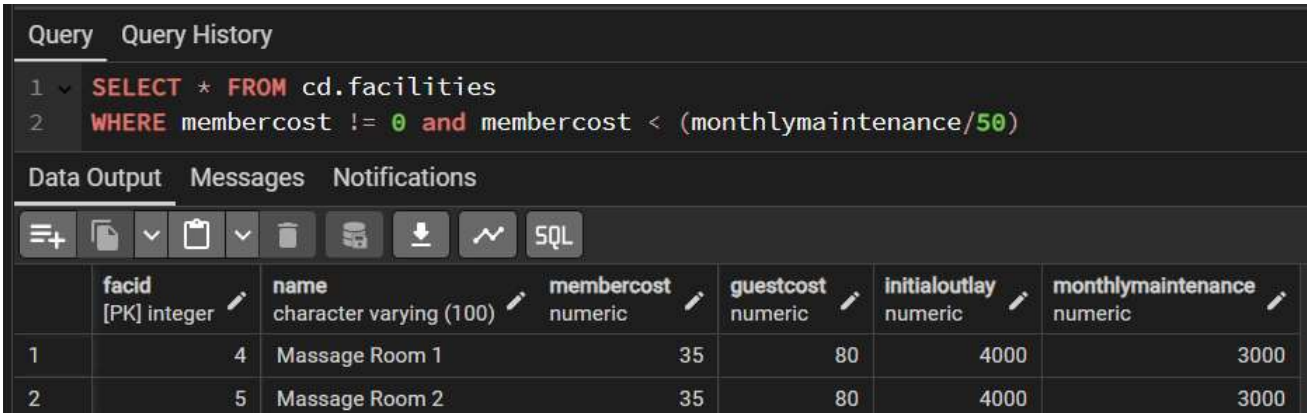
```
SELECT * FROM cd.facilities
WHERE membercost != 0
```

Data Output | Messages | Notifications

	facid [PK] integer	name character varying (100)	membercost numeric	guestcost numeric	initialoutlay numeric	monthlymaintenance numeric
1	0	Tennis Court 1	5	25	10000	200
2	1	Tennis Court 2	5	25	8000	200
3	4	Massage Room 1	35	80	4000	3000
4	5	Massage Room 2	35	80	4000	3000
5	6	Squash Court	3.5	17.5	5000	80

Ques 4) How can you produce a list of facilities that charge a fee to members, and that fee is less than 1/50th of the monthly maintenance cost? Return the facid, facility name, member cost, and monthly maintenance of the facilities in question.

```
SELECT * FROM cd.facilities
WHERE membercost != 0 and membercost < (monthlymaintenance/50)
```



Query | Query History

```
SELECT * FROM cd.facilities
WHERE membercost != 0 and membercost < (monthlymaintenance/50)
```

Data Output | Messages | Notifications

	facid [PK] integer	name character varying (100)	membercost numeric	guestcost numeric	initialoutlay numeric	monthlymaintenance numeric
1	4	Massage Room 1	35	80	4000	3000
2	5	Massage Room 2	35	80	4000	3000

Ques 5) How can you produce a list of all facilities with the word 'Tennis' in their name?

```
SELECT * FROM cd.facilities
WHERE name LIKE '%Tennis%'
```

Query | Query History

```
SELECT * FROM cd.facilities
WHERE name LIKE '%Tennis%'
```

Data Output | Messages | Notifications

	facid [PK] integer	name character varying (100)	membercost numeric	guestcost numeric	initialoutlay numeric	monthlymaintenance numeric
1	0	Tennis Court 1	5	25	10000	200
2	1	Tennis Court 2	5	25	8000	200
3	3	Table Tennis	0	5	320	10

Ques 6) How can you retrieve the details of facilities with ID 1 and 5? Try to do it without using the OR operator.

```
SELECT * FROM cd.facilities
WHERE facid IN (1,5)
```

Query | Query History

```
SELECT * FROM cd.facilities
WHERE facid IN (1,5)
```

Data Output | Messages | Notifications

	facid [PK] integer	name character varying (100)	membercost numeric	guestcost numeric	initialoutlay numeric	monthlymaintenance numeric
1	1	Tennis Court 2	5	25	8000	200
2	5	Massage Room 2	35	80	4000	3000

Ques 7) How can you produce a list of members who joined after the start of September 2012? Return the memid, surname, firstname, and joindate of the members in question.

```
SELECT memid, surname, firstname,joindate  FROM cd.members
WHERE Date(joindate) >= '2012-09-01'
```

Query | Query History

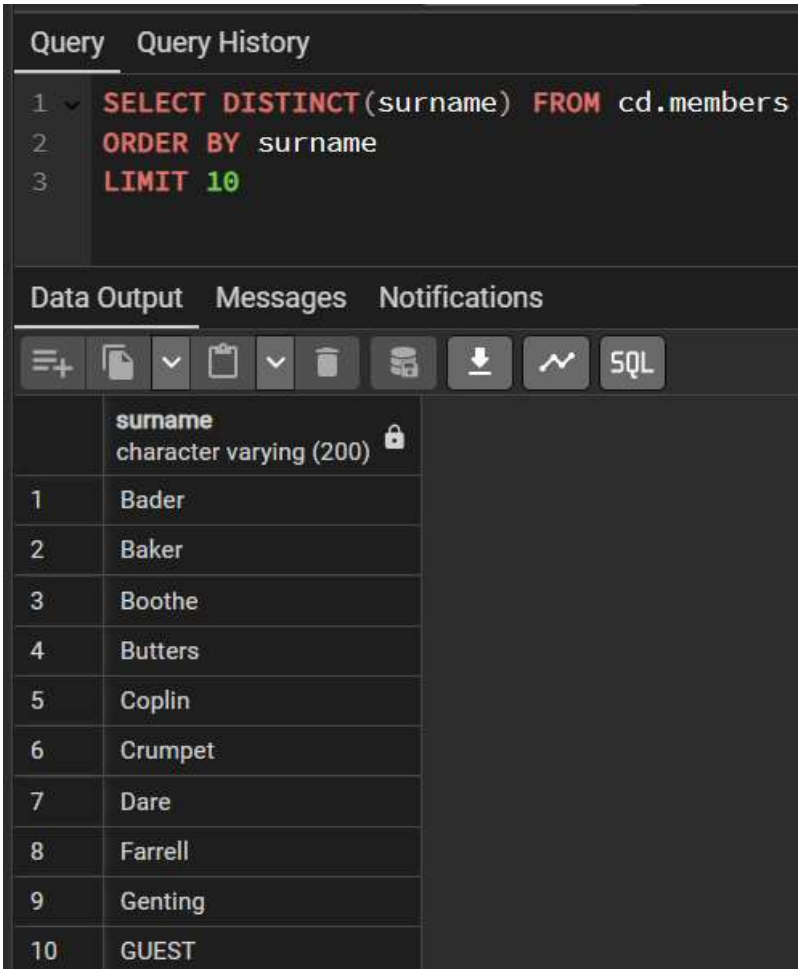
```
1  SELECT memid, surname, firstname,joindate  FROM cd.members
2  WHERE Date(joindate) >= '2012-09-01'
```

Data Output | Messages | Notifications

	memid [PK] integer	surname character varying (200)	firstname character varying (200)	joindate timestamp without time zone
1	24	Sarwin	Ramnaresh	2012-09-01 08:44:42
2	26	Jones	Douglas	2012-09-02 18:43:05
3	27	Rumney	Henrietta	2012-09-05 08:42:35
4	28	Farrell	David	2012-09-15 08:22:05
5	29	Worthington-Smyth	Henry	2012-09-17 12:27:15
6	30	Purview	Millicent	2012-09-18 19:04:01
7	33	Tupperware	Hyacinth	2012-09-18 19:32:05
8	35	Hunt	John	2012-09-19 11:32:45
9	36	Crumpet	Erica	2012-09-22 08:36:38
10	37	Smith	Darren	2012-09-26 18:08:45

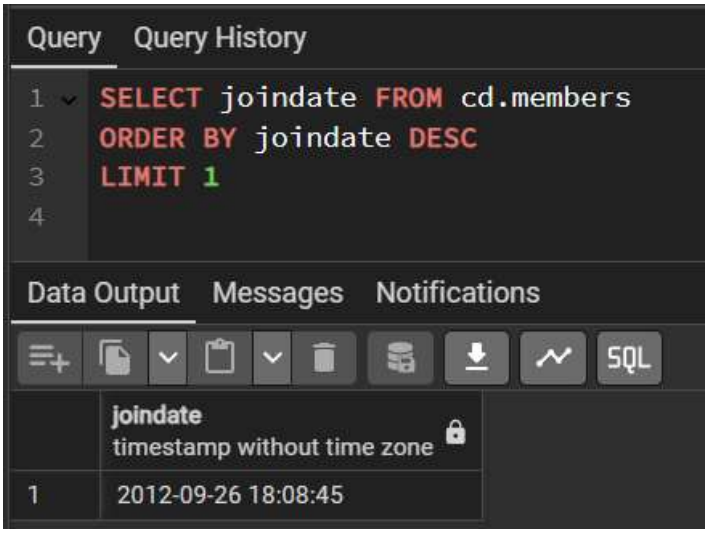
Ques 8) How can you produce an ordered list of the first 10 surnames in the members table? The list must not contain duplicates

```
SELECT DISTINCT(surname) FROM cd.members
ORDER BY surname
LIMIT 10
```



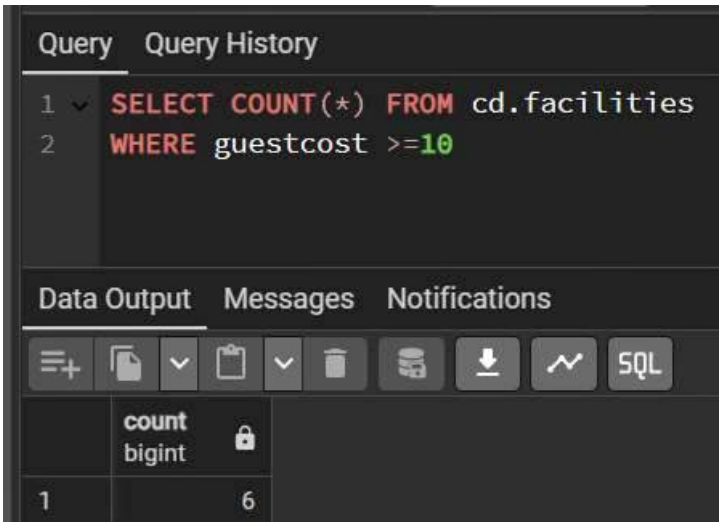
Ques 9) You'd like to get the signup date of your last member. How can you retrieve this information?

```
SELECT joindate FROM cd.members
ORDER BY joindate DESC
LIMIT 1
```



Ques 10) Produce a count of the number of facilities that have a cost to guests of 10 or more.

```
SELECT COUNT(*) FROM cd.facilities
WHERE guestcost >=10
```



Ques 11) Produce a list of the total number of slots booked per facility in the month of September 2012. Produce an output table consisting of facility id and slots, sorted by the number of slots.

```
SELECT facid, sum(slots) AS "Total Slots"
FROM cd.bookings
WHERE starttime >= '2012-09-01' AND starttime < '2012-10-01'
GROUP BY facid
ORDER BY SUM(slots) ASC;
```

Query | Query History

```
SELECT facid, sum(slots) AS "Total Slots"
FROM cd.bookings
WHERE starttime >= '2012-09-01' AND starttime < '2012-10-01'
GROUP BY facid
ORDER BY SUM(slots) ASC;
```

Data Output | Messages | Notifications

SQL

	facid integer	Total Slots bigint
1	5	122
2	3	422
3	7	426
4	8	471
5	6	540
6	2	570
7	1	588
8	0	591
9	4	648

Ques 12) Produce a list of facilities with more than 1000 slots booked. Produce an output table consisting of facility id and total slots, sorted by facility id.

```
SELECT facid, sum(slots) AS "Total Slots"
FROM cd.bookings
GROUP BY facid
HAVING SUM(slots) >1000
ORDER BY facid
```

WHERE clause:

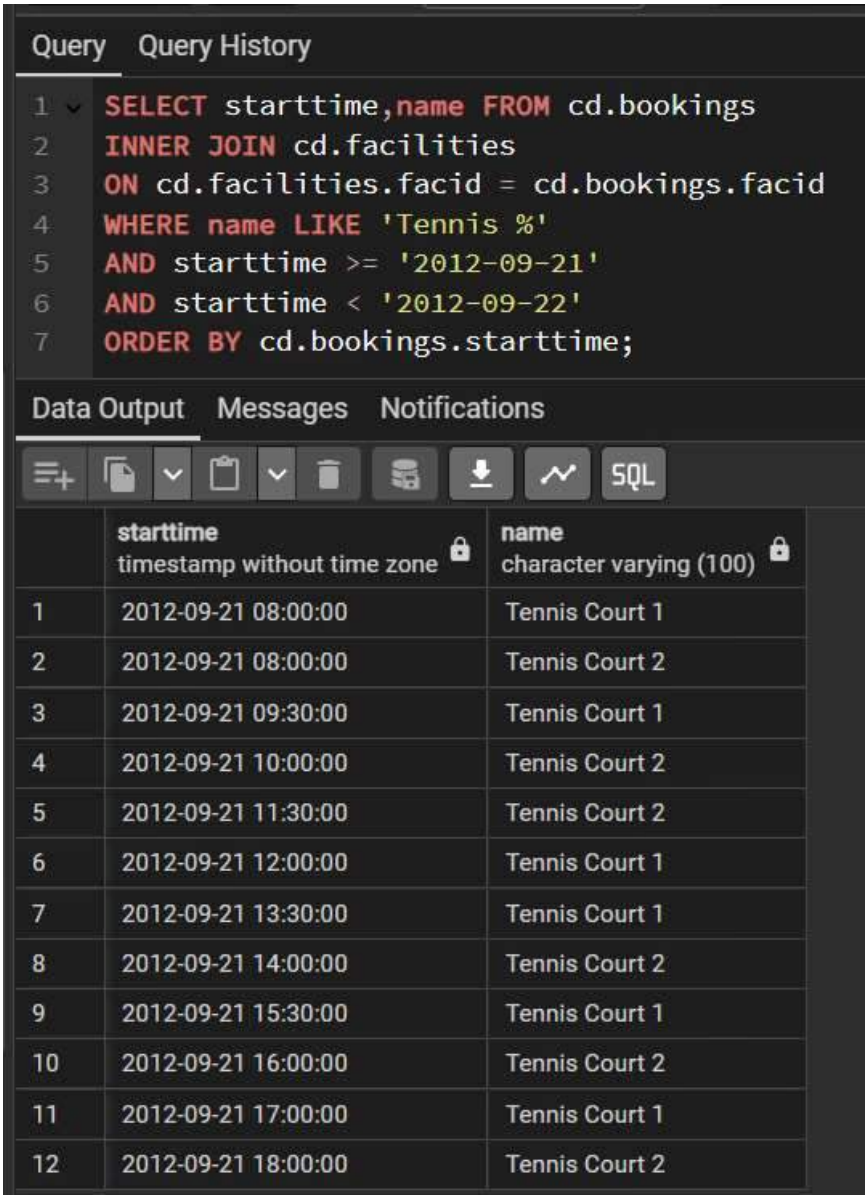
- It is used to filter rows before any aggregation occurs. You can use WHERE to filter based on column values but not with aggregate functions like SUM(), COUNT(), etc.

HAVING clause:

- It is used to filter rows after the aggregation has been performed. HAVING is specifically used with aggregate functions like SUM(), COUNT(), AVG(), etc. after the GROUP BY clause.
- Use HAVING after GROUP statememnt in a sql query

Ques 13) How can you produce a list of the start times for bookings for tennis courts, for the date '2012-09-21'? Return a list of start time and facility name pairings, ordered by the time.

```
SELECT starttime,name FROM cd.bookings
INNER JOIN cd.facilities
ON cd.facilities.facid = cd.bookings.facid
WHERE name LIKE 'Tennis %'
AND starttime >= '2012-09-21'
AND starttime < '2012-09-22'
ORDER BY cd.bookings.starttime;
```



	starttime timestamp without time zone	name character varying (100)
1	2012-09-21 08:00:00	Tennis Court 1
2	2012-09-21 08:00:00	Tennis Court 2
3	2012-09-21 09:30:00	Tennis Court 1
4	2012-09-21 10:00:00	Tennis Court 2
5	2012-09-21 11:30:00	Tennis Court 2
6	2012-09-21 12:00:00	Tennis Court 1
7	2012-09-21 13:30:00	Tennis Court 1
8	2012-09-21 14:00:00	Tennis Court 2
9	2012-09-21 15:30:00	Tennis Court 1
10	2012-09-21 16:00:00	Tennis Court 2
11	2012-09-21 17:00:00	Tennis Court 1
12	2012-09-21 18:00:00	Tennis Court 2

Ques 14) How can you produce a list of the start times for bookings by members named 'David Farrell'?

```
SELECT firstname,surname,starttime FROM cd.members
INNER JOIN cd. bookings
ON cd.members.memid = cd.bookings.memid
WHERE firstname LIKE 'David' AND surname LIKE 'Farrell'
```

Query | Query History

```
1  SELECT firstname,surname,starttime FROM cd.members
2  INNER JOIN cd. bookings
3  ON cd.members.memid = cd.bookings.memid
4  WHERE firstname LIKE 'David' AND surname LIKE 'Farrell'
```

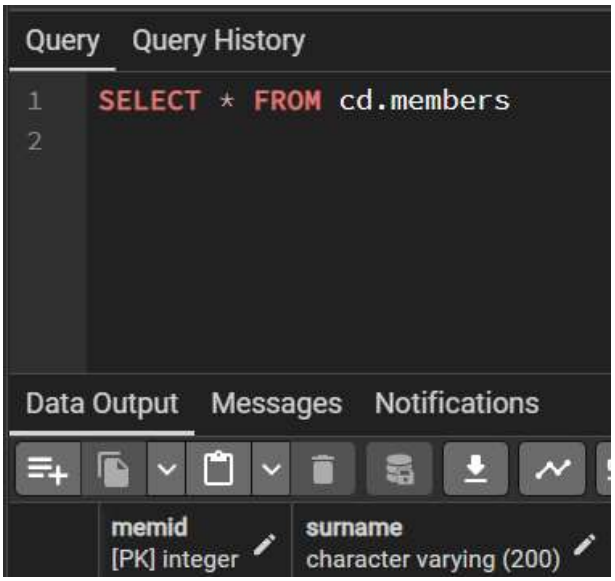
Data Output | Messages | Notifications

	firstname character varying (200)	surname character varying (200)	starttime timestamp without time zone
1	David	Farrell	2012-09-18 09:00:00
2	David	Farrell	2012-09-18 13:30:00
3	David	Farrell	2012-09-18 17:30:00
4	David	Farrell	2012-09-18 20:00:00

Creating Database and Tables

Primary key

- A primary key is a column or a group of columns used to identify a row uniquely in a table
- **Properties**:
  - Must be unique for each row (no duplicates).
  - Cannot contain `NULL` values (mandatory for every row).
  - A table can have only **one** primary key, but it can consist of multiple columns (composite key).
- Here It can be Seen `[PK] = Primary key`



Foreign key

- A column or set of columns in one table that refers to the primary key in another table.
- **Purpose**: To create a relationship between two tables and enforce referential integrity (ensure that data in one table corresponds to valid data in another table).
- **Properties**:
  - Can have duplicate values.
  - Can contain `NULL` values (unless constrained otherwise).
  - There can be multiple foreign keys in a table.
  - The foreign key value must match an existing value in the referenced primary key (or be `NULL` if allowed).

Constraints

A **constraint** in a database is a rule enforced on data columns in a table. Constraints help ensure the accuracy and integrity of the data by specifying conditions that data in the database must meet. They can be applied to one or more columns of a table during the table creation or modification process.

- **Primary key constraints**: Ensures that each row in a table is unique and cannot be `NULL` .

- **Foreign key constraints**: Ensures that a column's value matches the values in a related table, enforcing referential integrity.
- **Unique constraints**
- **Not Null**
- **Check constraint**: Ensures that a column's values meet a specified conditions
- **Refrences**: To constraint the value stored in the column that must exist in a column in another table.

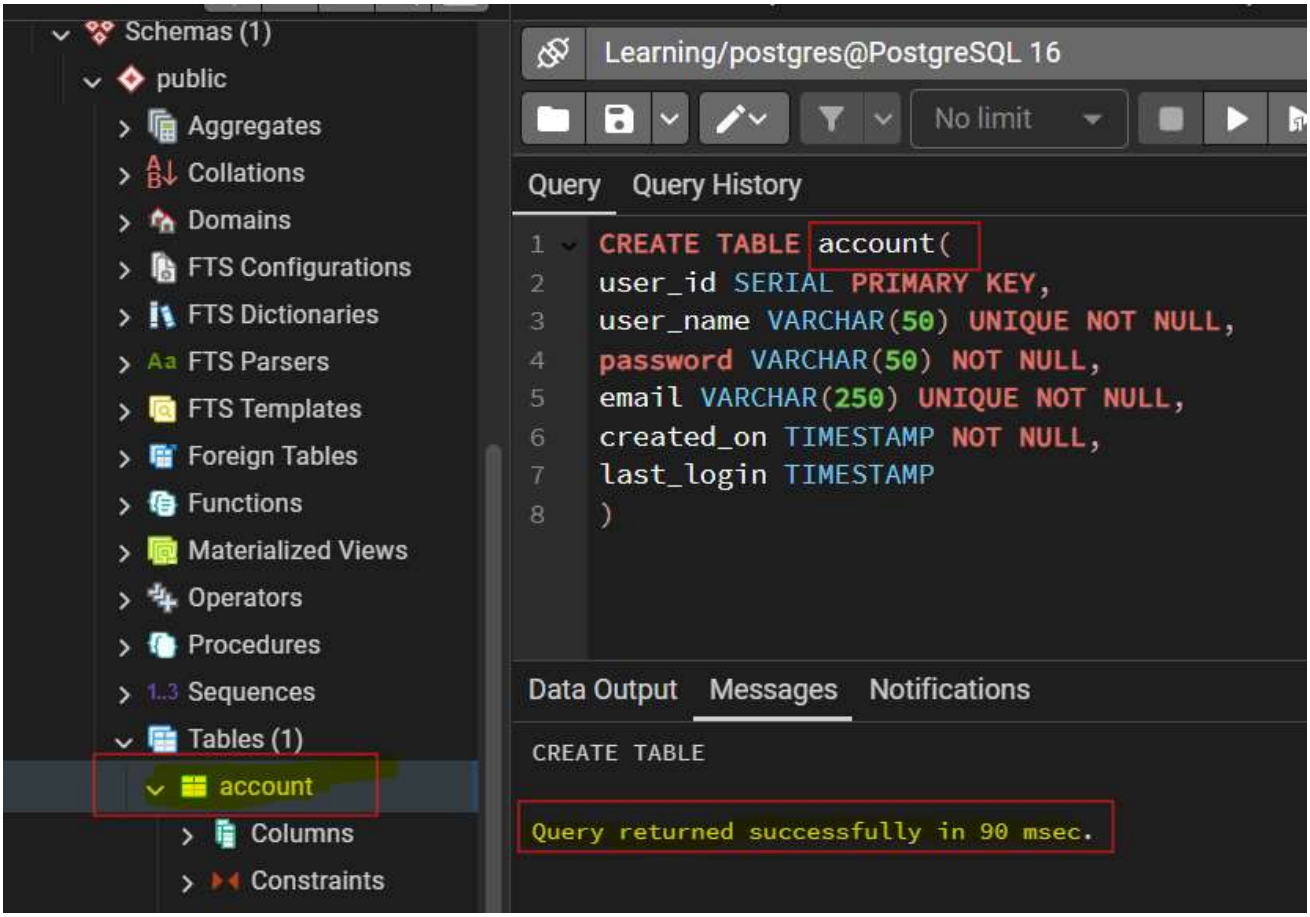
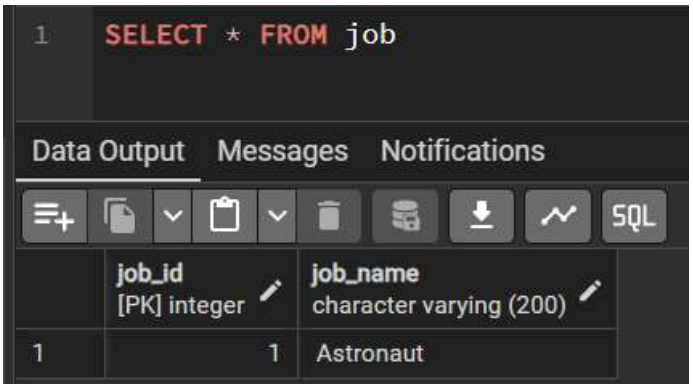
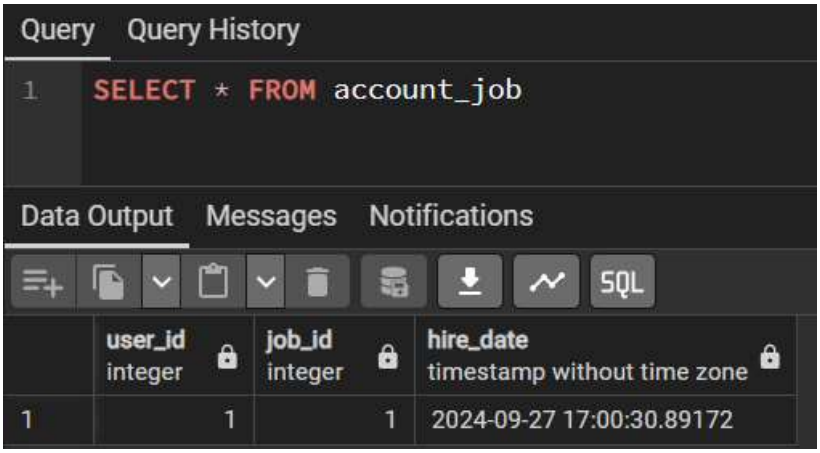
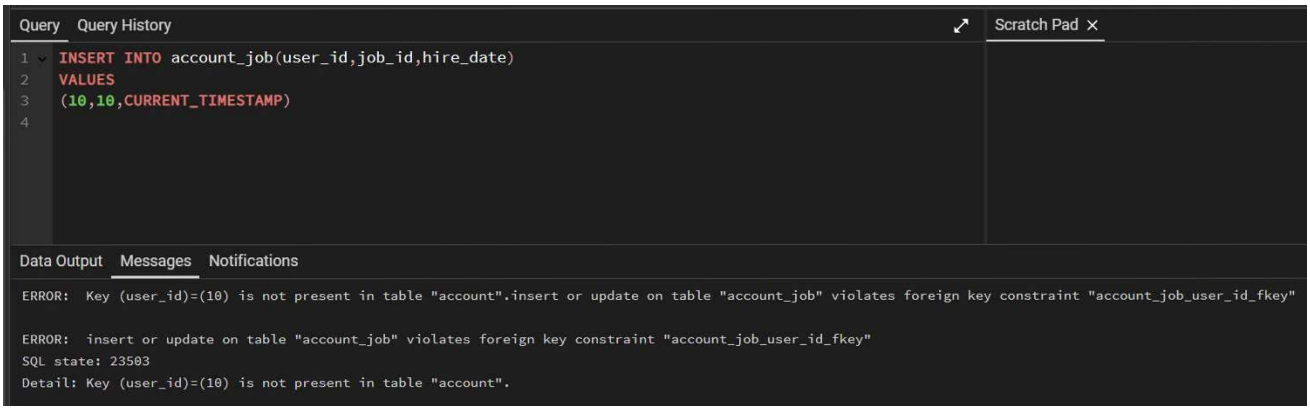
Create a Table

- ```
  CREATE TABLE table_name (
      column1 datatype constraint,
      column2 datatype constraint,
      column3 datatype constraint,
      ...
  );
  ```

```
CREATE TABLE Employees (
    employee_id INT PRIMARY KEY,        -- Primary key constraint
    first_name VARCHAR(50) NOT NULL,    -- Not null constraint
    last_name VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE,          -- Unique constraint
    hire_date DATE DEFAULT CURRENT_DATE -- Default constraint
);
```

Common Datatypes in SQl

Here's a table of the most common SQL data types, along with their syntax and brief descriptions:

Data Type	Syntax	Description
Integer Types		
`INT / INTEGER`	`INT or INTEGER`	Stores whole numbers (positive, negative, or zero).
`SMALLINT`	`SMALLINT`	Stores small-range integers.
`BIGINT`	`BIGINT`	Stores very large whole numbers.
Decimal & Floating-Point Types		
`DECIMAL(p,s)`	`DECIMAL(p,s)`	Fixed precision and scale numbers. `p` is precision (total digits), `s` is scale (digits after decimal).
`NUMERIC(p,s)`	`NUMERIC(p,s)`	Same as `DECIMAL`, often used interchangeably.
`FLOAT`	`FLOAT or FLOAT(n)`	Stores floating-point numbers, where `n` specifies the precision.
`REAL`	`REAL`	Stores approximate floating-point values with lower precision than `FLOAT`.
```

Data Type	Syntax	Description
String Types		
`CHAR(n)`	`CHAR(n)`	Fixed-length character string of length `n`.
`VARCHAR(n)`	`VARCHAR(n)`	Variable-length character string of up to `n` characters.
`TEXT`	`TEXT`	Stores long strings of variable length.
Date/Time Types		
`DATE`	`DATE`	Stores date values (year, month, day).
`TIME`	`TIME`	Stores time values (hours, minutes, seconds).
`DATETIME`	`DATETIME`	Stores both date and time information.
`TIMESTAMP`	`TIMESTAMP`	Stores a timestamp (date and time) with time zone support in some systems.
Boolean Type		
`BOOLEAN`	`BOOLEAN`	Stores `TRUE`, `FALSE`, or `NULL`.
Binary Types		
`BINARY(n)`	`BINARY(n)`	Fixed-length binary data of length `n`.
`VARBINARY(n)`	`VARBINARY(n)`	Variable-length binary data of up to `n` bytes.
Other Types		
`BLOB`	`BLOB`	Stores binary large objects (e.g., images, audio).
`ENUM`	`ENUM('value1', 'value2')`	A string object that can only have one value, chosen from a list of values.

SERIAL

In SQL, the `SERIAL` data type is commonly used to create auto-incrementing integer columns, often used for primary keys. It automatically generates unique values for new rows without manually specifying a value.

- `SERIAL` is not a standard SQL data type but is specific to databases like **PostgreSQL**
- `SERIAL` implicitly creates a sequence and associates it with the column.

```
CREATE TABLE Employees (
    employee_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50)
);
```

Note: If the `SERIAL` data type is used and then if we delete any column then we will see a break in sequence of row number [This also true for Failed attempts in adding rows]

Task 1: Create a Table

```
CREATE TABLE account(
user_id SERIAL PRIMARY KEY,
user_name VARCHAR(50) UNIQUE NOT NULL,
password VARCHAR(50) NOT NULL,
email VARCHAR(250) UNIQUE NOT NULL,
created_on TIMESTAMP NOT NULL,
last_login TIMESTAMP
)
```




```
CREATE TABLE job(
job_id SERIAL PRIMARY KEY,
job_name VARCHAR(200) UNIQUE NOT NULL
);
```

```
CREATE TABLE account_job(
user_id INTEGER REFERENCES account(user_id),
job_id INTEGER REFERENCES job(job_id),
hire_date TIMESTAMP
);
```

we have created 3 Table

Insert Information in Tables

To insert rows into a table, you can use the `INSERT INTO` statement in SQL. Here's a general guide to inserting multiple rows into a table:

```
INSERT INTO account(user_name,password,email,created_on)
VALUES
('Jose','password','jose@mail.com',CURRENT_TIMESTAMP)
```

Note: user_id is automatically created for us

```
INSERT INTO job(job_name)
VALUES
('Astronaut')
```




```
INSERT INTO account_job(user_id,job_id,hire_date)
VALUES
(1,1,CURRENT_TIMESTAMP)
```



Note: Now we will try to add someone whose user_id does not exist

- This will violate the foreign key constraint

- ```
  CREATE TABLE account_job(
  user_id INTEGER REFERENCES account(user_id),
  job_id INTEGER REFERENCES job(job_id),
  hire_date TIMESTAMP
  );
  ```

- we can see job_id is a refrenced to Table "Job" since there no such `job.job_id` we cannot add that to account_job



Update

The `UPDATE` keyword in SQL is used to modify existing records in a table. You can update one or more columns based on a condition.

- we can either normally update a tabel row or we can use other table to update our current table

```
UPDATE Employees
SET hire_date = '2023-10-01'
WHERE hire_date < '2023-01-01';
```

```
UPDATE Employees
SET department_name = Departments.department_name
FROM Departments
WHERE Employees.department_id = Departments.department_id;
```

Returning

In SQL, the `RETURNING` clause is used to return the updated, inserted, or deleted rows immediately after an `INSERT`, `UPDATE`, or `DELETE` operation. It's particularly useful when you need to retrieve values from a table after making changes, such as generated `SERIAL` values or updated fields

```
UPDATE Employees
SET hire_date = '2023-10-01'
WHERE employee_id = 1
RETURNING employee_id, hire_date;
```

DELETE
```

In SQL, the `DELETE` statement is used to remove rows from a table. You can delete specific rows based on a condition, or, if you omit the condition, you can delete all rows from a table.

```
DELETE FROM table_name
WHERE condition;
```

ALTER

The `ALTER` statement in SQL is used to modify the structure of an existing table. You can use it to add, modify, rename, or drop columns, or to change other properties of a table.

Operation	SQL Syntax	Example
Add a Column	`ALTER TABLE table_name ADD column_name datatype;`	`ALTER TABLE Employees ADD phone_number VARCHAR(15);`
Modify a Column	MySQL: `ALTER TABLE table_name MODIFY column_name new_datatype;` PostgreSQL: `ALTER TABLE table_name ALTER COLUMN column_name TYPE new_datatype;`	MySQL: `ALTER TABLE Employees MODIFY phone_number VARCHAR(20);` PostgreSQL: `ALTER TABLE Employees ALTER COLUMN phone_number TYPE VARCHAR(20);`
Rename a Column	`ALTER TABLE table_name RENAME COLUMN old_column_name TO new_column_name;`	`ALTER TABLE Employees RENAME COLUMN phone_number TO contact_number;`
Drop a Column	`ALTER TABLE table_name DROP COLUMN column_name;`	`ALTER TABLE Employees DROP COLUMN contact_number;`
Rename a Table	`ALTER TABLE old_table_name RENAME TO new_table_name;`	`ALTER TABLE Employees RENAME TO Staff;`

In SQL, the `ALTER` statement can also be used with the `SET` keyword to modify specific attributes of columns or tables, such as setting default values or constraints. Here are common use cases for `ALTER` with `SET`.

Common `ALTER` + `SET` Operations:

Operation	SQL Syntax	Example
Set a Default Value	`ALTER TABLE table_name ALTER COLUMN column_name SET DEFAULT default_value;`	`ALTER TABLE Employees ALTER COLUMN hire_date SET DEFAULT CURRENT_DATE;`
Remove a Default Value	`ALTER TABLE table_name ALTER COLUMN column_name DROP DEFAULT;`	`ALTER TABLE Employees ALTER COLUMN hire_date DROP DEFAULT;`
Set a Column to NOT NULL	`ALTER TABLE table_name ALTER COLUMN column_name SET NOT NULL;`	`ALTER TABLE Employees ALTER COLUMN last_name SET NOT NULL;`
Remove NOT NULL Constraint	`ALTER TABLE table_name ALTER COLUMN column_name DROP NOT NULL;`	`ALTER TABLE Employees ALTER COLUMN last_name DROP NOT NULL;`
Set a Table to Read Only (PostgreSQL)	`ALTER TABLE table_name SET READ ONLY;`	`ALTER TABLE Employees SET READ ONLY;`
Set a Table to Read Write (PostgreSQL)	`ALTER TABLE table_name SET READ WRITE;`	`ALTER TABLE Employees SET READ WRITE;`

```sql
ALTER TABLE table_name
ADD CONSTRAINT constraint_name PRIMARY KEY (column_name);
```

```sql
ALTER TABLE Employees
ADD CONSTRAINT pk_employee_id PRIMARY KEY (employee_id);
```

CHECK

- The CHECK constraint is a powerful feature to enforce data integrity in your tables.
- The **condition** in the CHECK constraint can include one or more columns and can involve various operators such as comparison (`>` , `<` , `=` , etc.) and logical operators (`AND` , `OR`).

```sql
CREATE TABLE Employees (
    employee_id SERIAL PRIMARY KEY,
    age INT,
    parent_age INT,
    CONSTRAINT chk_age CHECK (parent_age > age)
    -- Ensures parent_age is greater than age
);
```

Assessment Test - 3

Task-1

Create a new database called "School" this database should have two tables: **teachers** and **students.**

The **students** table should have columns for student_id, first_name,last_name, homeroom_number, phone,email, and graduation year.

The **teachers** table should have columns for teacher_id, first_name, last_name,

homeroom_number, department, email, and phone.

The constraints are mostly up to you, but your table constraints do have to consider the following:

1. We must have a phone number to contact students in case of an emergency.
2. We must have ids as the primary key of the tables
3. Phone numbers and emails must be unique to the individual.

Once you've made the tables, insert a student named Mark Watney (student_id=1) who has a phone number of 777-555-1234 and doesn't have an email. He graduates in 2035 and has 5 as a homeroom number.

Then insert a teacher names Jonas Salk (teacher_id = 1) who as a homeroom number of 5 and is from the Biology department. His contact info is: jsalk@school.org and a phone number of 777-555-4321.

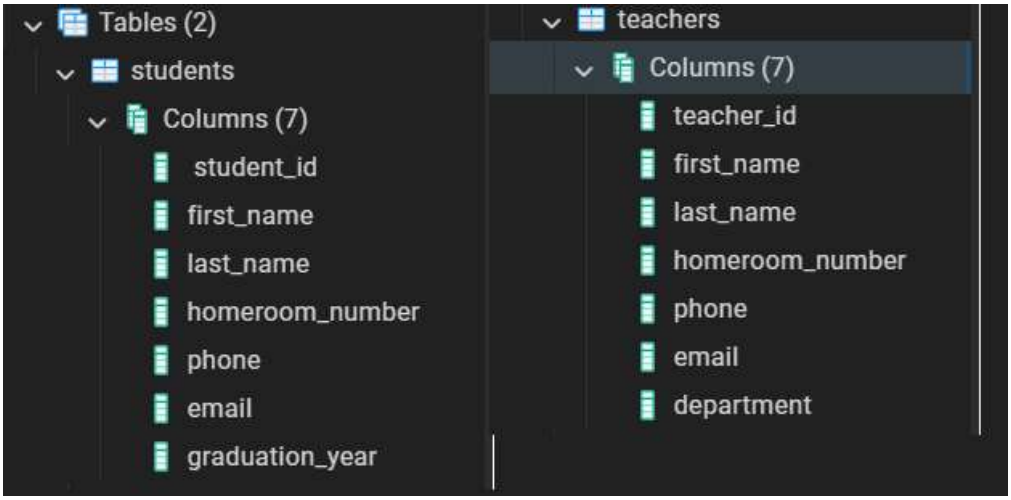
Solution

- Creating the student table

```sql
CREATE TABLE student (
    student_id SERIAL PRIMARY KEY,      -- Auto-incrementing primary key
    first_name VARCHAR(50),             -- First name with a max length of 50 characters
    last_name VARCHAR(50),              -- Last name with a max length of 50 characters
    homeroom_number INT,                -- Integer column for homeroom number
    phone VARCHAR(200) NOT NULL UNIQUE, -- Phone number, cannot be NULL, must be unique
    email VARCHAR(200) UNIQUE,          -- Email, must be unique but can be NULL
    graduation_year INT                 -- Integer for graduation year
);
```

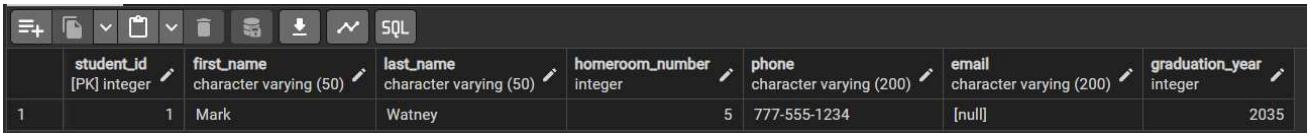
- Creating the teachers Table

```sql
CREATE TABLE teachers (
    teacher_id SERIAL PRIMARY KEY,      -- Auto-incrementing primary key
    first_name VARCHAR(50),             -- First name with a max length of 50 characters
    last_name VARCHAR(50),              -- Last name with a max length of 50 characters
    homeroom_number INT,                -- Integer column for homeroom number
    phone VARCHAR(200) NOT NULL UNIQUE, -- Phone number, cannot be NULL, must be unique
    email VARCHAR(200) UNIQUE,          -- Email, must be unique but can be NULL
    department VARCHAR(50)              -- Department name with a max length of 50 characters
);
```



- Adding the student named Mark Watney (student_id=1) who has a phone number of 777-555-1234 and doesn't have an email. He graduates in 2035 and has 5 as a homeroom numbe

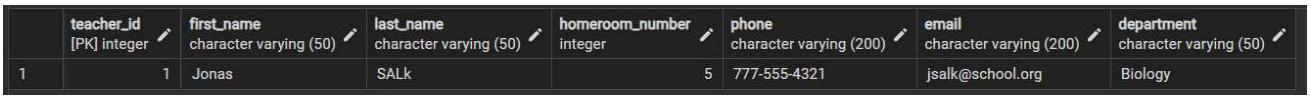
```
INSERT INTO students (
first_name, last_name, phone, graduation_year, homeroom_number)
VALUES ('Mark', 'Watney', '777-555-1234', 2035, 5);
```



	student_id [PK] integer	first_name character varying (50)	last_name character varying (50)	homeroom_number integer	phone character varying (200)	email character varying (200)	graduation_year integer
1	1	Mark	Watney	5	777-555-1234	[null]	2035

- Then insert a teacher names Jonas Salk (teacher_id = 1) who as a homeroom number of 5 and is from the Biology department. His contact info is: jsalk@school.org and a phone number of 777-555-4321.

```
INSERT INTO teachers (
first_name, last_name, homeroom_number,department,phone,email)
VALUES (
'Jonas', 'SALk',5,'Biology','777-555-4321','jsalk@school.org');
```



	teacher_id [PK] integer	first_name character varying (50)	last_name character varying (50)	homeroom_number integer	phone character varying (200)	email character varying (200)	department character varying (50)
1	1	Jonas	SALk	5	777-555-4321	jsalk@school.org	Biology

Conditional Expessions and Operators

CASE

- A **SQL case statement** is used to execute a series of conditional expressions and returns one of the specified results when a condition is true. It is often used to transform data or apply conditional logic directly within SQL queries.

```
  SELECT
    CASE
      WHEN condition1 THEN result1
      WHEN condition2 THEN result2
      ...
      ELSE resultN
    END AS alias_name
  FROM table_name;
```

```
SELECT
  employee_name,
  salary,
  CASE
    WHEN salary > 50000 THEN 'High Salary'
    WHEN salary BETWEEN 30000 AND 50000 THEN 'Medium Salary'
    ELSE 'Low Salary'
  END AS salary_category
FROM employees;
```

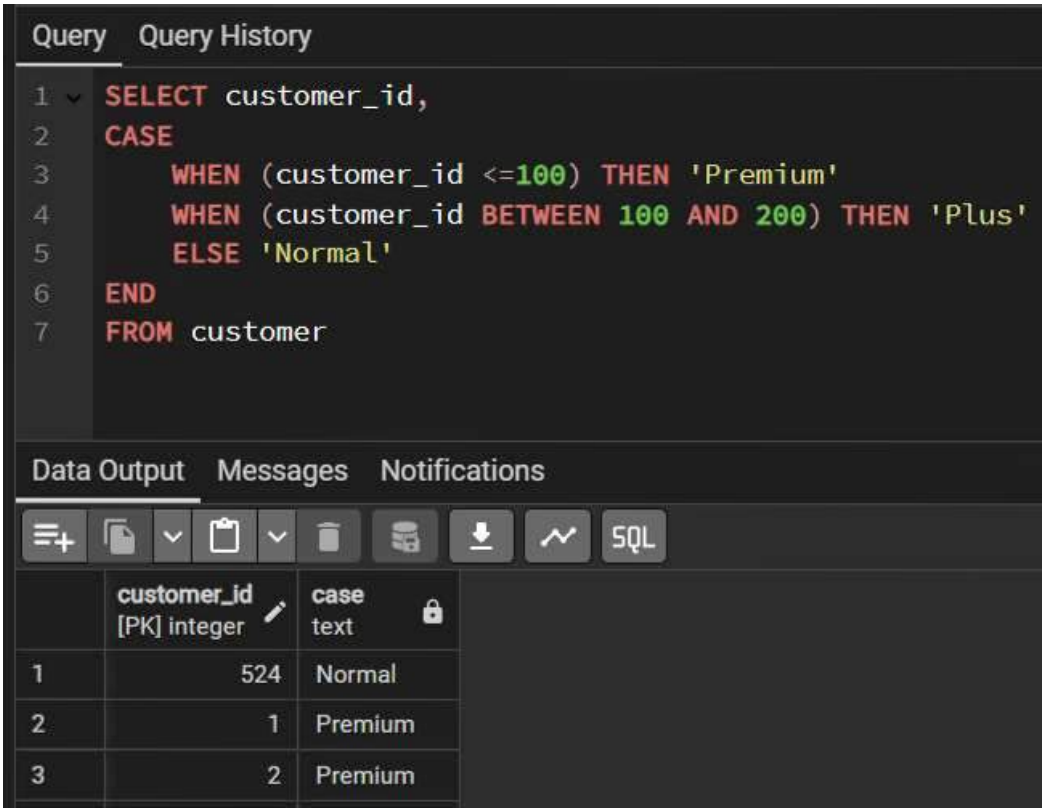
- You can use `CASE` in `SELECT`, `WHERE`, `ORDER BY`, or even `JOIN` clauses, providing a flexible way to handle conditional logic

CASE Expression

A **`CASE Expression`** is the **complete evaluation** that happens as part of a SQL query using the `CASE` structure. It's the term used to describe the result of the `CASE` operation in a query.

```
CASE expression -- [Expression here could be a cloumn_name]
  WHEN value1 THEN result1
  WHEN value2 THEN result2
  ...
  ELSE resultN
END
```

```
-- CASE
SELECT customer_id,
CASE
        WHEN (customer_id <=100) THEN 'Premium'
        WHEN (customer_id BETWEEN 100 AND 200) THEN 'Plus'
        ELSE 'Normal'
END
FROM customer
```




```sql
-- CASE EXPRESSION
SELECT customer_id,
CASE customer_id
        WHEN 2 THEN 'Winner'
        WHEN 5 THEN 'Runner-up'
        ELSE 'Normal'
END AS Lottery_winner
FROM customer
```

Query | Query History

```sql
SELECT customer_id,
CASE customer_id
    WHEN 2 THEN 'Winner'
    WHEN 5 THEN 'Runner-up'
    ELSE 'Normal'
END AS Lottery_winner
FROM customer
```

Data Output | Messages | Notifications

	customer_id [PK] integer	lottery_winner text
1	524	Normal
2	1	Normal
3	2	Winner
4	3	Normal
5	4	Normal
6	5	Runner-up

- We can also use CASE statment to count a particular value is repeated how many time

```sql
SELECT
SUM(CASE rental_rate
        WHEN 0.99 THEN 1
        ELSE 0
END) AS number_of_bargin
FROM film
```

Query | Query History

```sql
SELECT
SUM(CASE rental_rate
    WHEN 0.99 THEN 1
    ELSE 0
END) AS number_of_bargin
FROM film
```

Data Output | Messages | Notifications

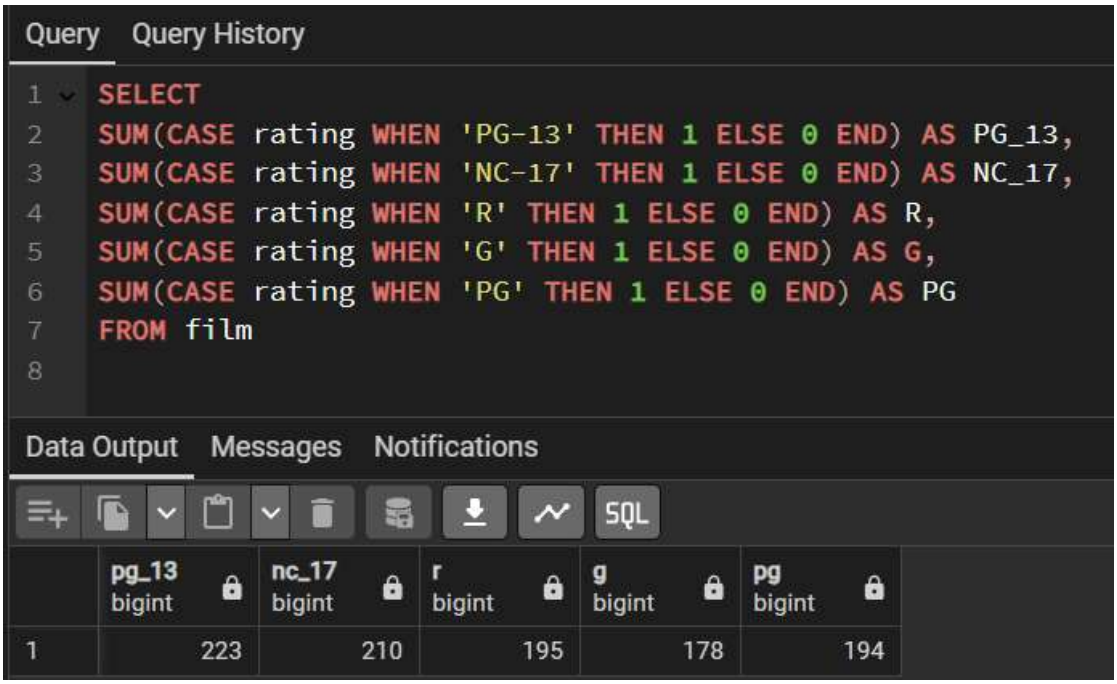
	number_of_bargin bigint
1	341

Here we are calculating how many movies are 0.99 the shop has

⚔️ CASE Challenge

Ques) we want to know and compare the various amounts of films we have per movie rating. Use CASE and dvd_rental Database

```
SELECT
SUM(CASE rating WHEN 'PG-13' THEN 1 ELSE 0 END) AS PG_13,
SUM(CASE rating WHEN 'NC-17' THEN 1 ELSE 0 END) AS NC_17,
SUM(CASE rating WHEN 'R' THEN 1 ELSE 0 END) AS R,
SUM(CASE rating WHEN 'G' THEN 1 ELSE 0 END) AS G,
SUM(CASE rating WHEN 'PG' THEN 1 ELSE 0 END) AS PG
FROM film
```



COALESCE

The `COALESCE` function in SQL is used to return the first non-null value from a list of expressions. It's particularly useful for handling null values and providing default values when dealing with potentially missing data.

- It's commonly used for ensuring that queries return meaningful data when nulls are present. [If you want to perform operation on a column but there are some NULL in it then you can use COALESCE to handle those NULL]

```
SELECT
  employee_id,
  COALESCE(email_address, personal_email, 'No Email Provided') AS email
FROM employees;
```

CAST

The `CAST` function in SQL is used to convert an expression from one data type to another. It is particularly useful when you need to ensure that data types match for operations, comparisons, or for formatting purposes

```sql
SELECT
  CAST(age AS INTEGER) AS age_integer
FROM users;
```

```sql
SELECT
  '123'::INTEGER AS converted_integer;
```

Here's a table of the most common data types in PostgreSQL along with their syntax for using the `CAST` function:

Data Type	CAST Syntax	Example
INTEGER	`CAST(value AS INTEGER)`	`SELECT CAST('42' AS INTEGER) AS integer_value;`
BIGINT	`CAST(value AS BIGINT)`	`SELECT CAST('123456789012345' AS BIGINT) AS big_integer_value;`
SMALLINT	`CAST(value AS SMALLINT)`	`SELECT CAST('30000' AS SMALLINT) AS small_integer_value;`
DECIMAL/NUMERIC	`CAST(value AS DECIMAL(precision, scale))`	`SELECT CAST('123.456' AS DECIMAL(5, 3)) AS decimal_value;`
REAL	`CAST(value AS REAL)`	`SELECT CAST('3.14' AS REAL) AS real_value;`
DOUBLE PRECISION	`CAST(value AS DOUBLE PRECISION)`	`SELECT CAST('3.14159' AS DOUBLE PRECISION) AS double_value;`
VARCHAR	`CAST(value AS VARCHAR(length))`	`SELECT CAST(42 AS VARCHAR(10)) AS varchar_value;`
TEXT	`CAST(value AS TEXT)`	`SELECT CAST(123 AS TEXT) AS text_value;`
CHAR	`CAST(value AS CHAR(length))`	`SELECT CAST('A' AS CHAR(1)) AS char_value;`
DATE	`CAST(value AS DATE)`	`SELECT CAST('2024-09-28' AS DATE) AS date_value;`
TIME	`CAST(value AS TIME)`	`SELECT CAST('12:34:56' AS TIME) AS time_value;`
TIMESTAMP	`CAST(value AS TIMESTAMP)`	`SELECT CAST('2024-09-28 10:30:00' AS TIMESTAMP) AS timestamp_value;`
BOOLEAN	`CAST(value AS BOOLEAN)`	`SELECT CAST('true' AS BOOLEAN) AS boolean_value;`

NULLIF

- The `NULLIF` function in SQL is used to return `NULL` if two expressions are equal. If the expressions are not equal, it returns the first expression.
- This function can be particularly useful for handling cases where you want to avoid division by zero or when you want to substitute a specific value with `NULL`.

```sql
SELECT
    salary / NULLIF(bonus, 0) AS adjusted_salary
FROM employees;
-- In this example, if bonus is 0,
-- the division would return NULL
-- instead of causing a division by zero error.


-- Conditional Statment using NULLIF
SELECT
    employee_id,
    CASE
        WHEN NULLIF(bonus, 0) IS NULL THEN 'No Bonus'
        ELSE 'Bonus Available'
    END AS bonus_status
FROM employees;
```

VIEWS

In SQL, a **view** is a virtual table that provides a way to present data from one or more tables in a structured format. Views can simplify complex queries, enhance security by restricting access to certain data, and provide a consistent interface to the underlying data.

```sql
-- Let say we have this below query
SELECT first_name,last_name,address FROM customer
INNER JOIN address
ON customer.address_id = address.address_id
```

so instead of writing the query every time we can create a VIEW for it which can be called later directly

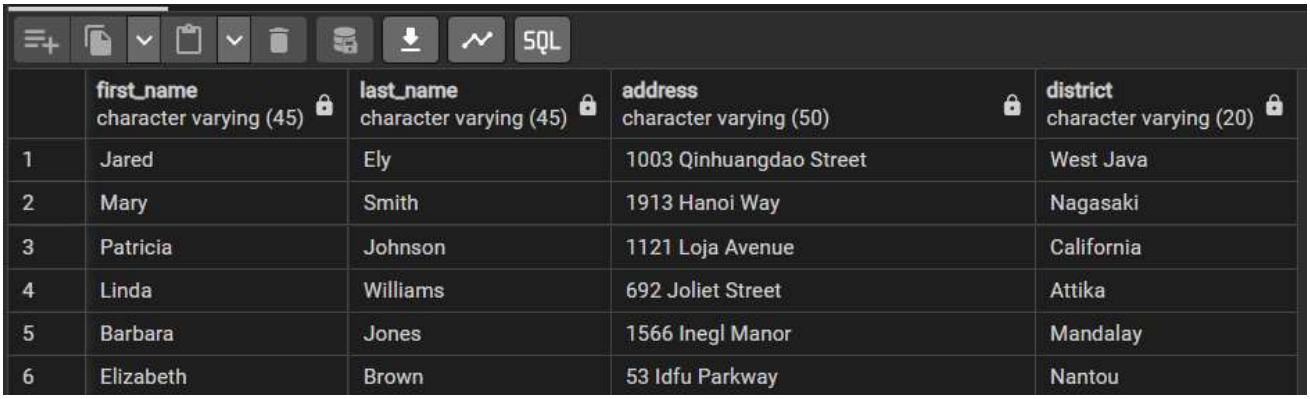
```sql
CREATE VIEW customer_info AS
SELECT first_name,last_name,address FROM customer
INNER JOIN address
ON customer.address_id = address.address_id


-- To call it next time
SELECT * FROM customer_info
```

	first_name character varying (45)	last_name character varying (45)	address character varying (50)
1	Jared	Ely	1003 Qinhuangdao Street
2	Mary	Smith	1913 Hanoi Way
3	Patricia	Johnson	1121 Loja Avenue
4	Linda	Williams	692 Joliet Street

- We can also REPLACE [edit] the VIEW

```sql
CREATE OR REPLACE VIEW customer_info AS
SELECT first_name,last_name,address,district FROM customer
INNER JOIN address
ON customer.address_id = address.address_id
```

```
-- we have added a new column "district"
```



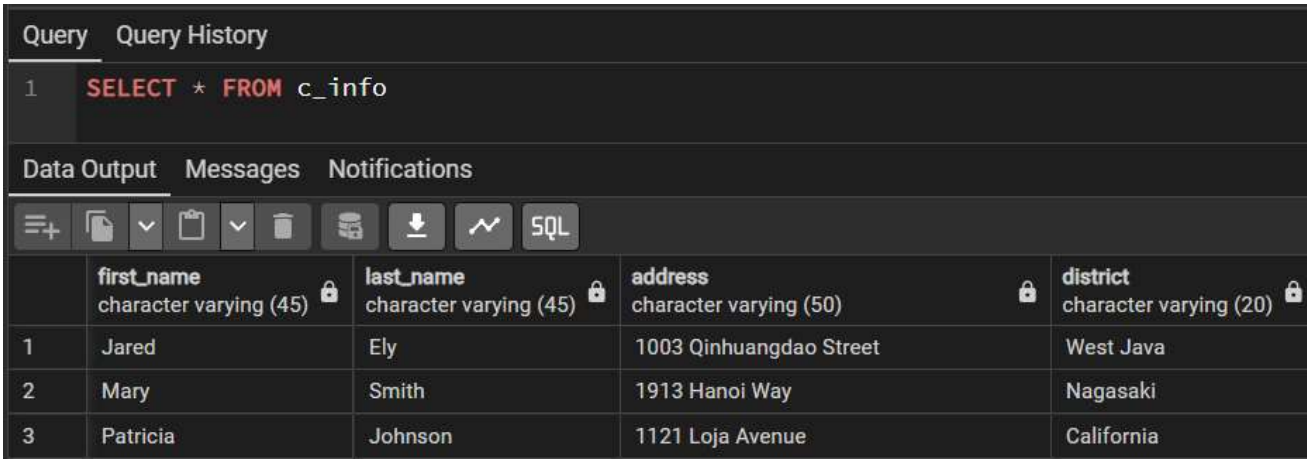
	first_name character varying (45)	last_name character varying (45)	address character varying (50)	district character varying (20)
1	Jared	Ely	1003 Qinhuangdao Street	West Java
2	Mary	Smith	1913 Hanoi Way	Nagasaki
3	Patricia	Johnson	1121 Loja Avenue	California
4	Linda	Williams	692 Joliet Street	Attika
5	Barbara	Jones	1566 Inegl Manor	Mandalay
6	Elizabeth	Brown	53 Idfu Parkway	Nantou

- To delete your view

```
DROP VIEW IF EXIST customer_info
```

- To change the name of VIEW

```
ALTER VIEW customer_info RENAME to c_info
```



	first_name character varying (45)	last_name character varying (45)	address character varying (50)	district character varying (20)
1	Jared	Ely	1003 Qinhuangdao Street	West Java
2	Mary	Smith	1913 Hanoi Way	Nagasaki
3	Patricia	Johnson	1121 Loja Avenue	California

ROW_NUMBER()

The `ROW_NUMBER()` function is a window function in SQL that assigns a unique sequential integer to rows within a partition of a result set. This function is often used for tasks such as ranking data, paginating results, or identifying duplicates.

```
ROW_NUMBER()
OVER (PARTITION BY column1, column2, ... ORDER BY column_name [ASC|DESC])
AS ranked_columns
```

- The `PARTITION BY` clause divides the result set into partitions to which the `ROW_NUMBER()` function is applied. [optional clause]
    - It defines the groups of rows to which the `ROW_NUMBER()` function is applied.

- The `ORDER BY` clause within the `OVER` clause defines the order in which the rows are numbered. [mandatory clause]

Example

```
SELECT
    employee_id,
    first_name,
    last_name,
    department_id,
    salary,
    ROW_NUMBER() OVER (PARTITION BY department_id ORDER BY salary DESC) AS salary_rank
FROM employees;
```

The output will be:

employee_id	first_name	last_name	department_id	salary	salary_rank
2	Jane	Smith	1	60000	1
1	John	Doe	1	50000	2
5	Chris	Davis	1	45000	3
4	Mike	Brown	2	65000	1
3	Emily	Johnson	2	55000	2

Practice Question

Select the 3rd highest salary of each department and if the department has less than 3 memeber than return the lowest salary

```
WITH RankedSalaries AS (
    SELECT
        department_id,
        salary,
        ROW_NUMBER() OVER (PARTITION BY department_id ORDER BY salary DESC) AS salary_rank,
        COUNT(*) OVER (PARTITION BY department_id) AS member_count
    FROM employees
)

SELECT
    department_id,
    CASE
        WHEN member_count < 3 THEN MIN(salary)
        ELSE MAX(CASE WHEN salary_rank = 3 THEN salary END)
    END AS salary_value
FROM RankedSalaries
GROUP BY department_id, member_count;
```

The results are grouped by `department_id` and `member_count` to ensure the query returns one row per department.