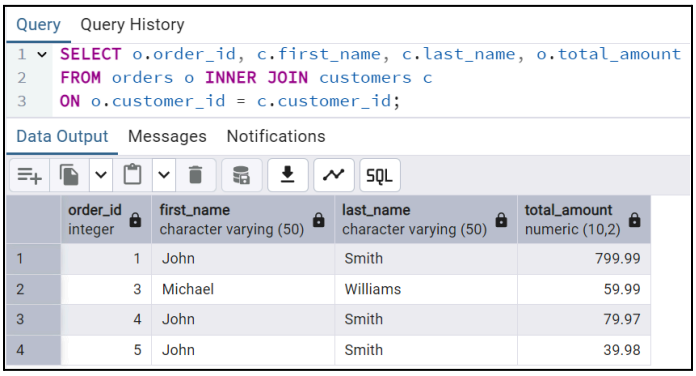
TASK-5

INNER JOIN: returns only the rows where there is a match in both tables based on the specified condition. It excludes all rows that do not have corresponding entries in both joined tables. This is the most commonly used join when the goal is to fetch data that has a direct relationship between two tables. For example, joining customers with their orders will only show those who have actually placed orders.



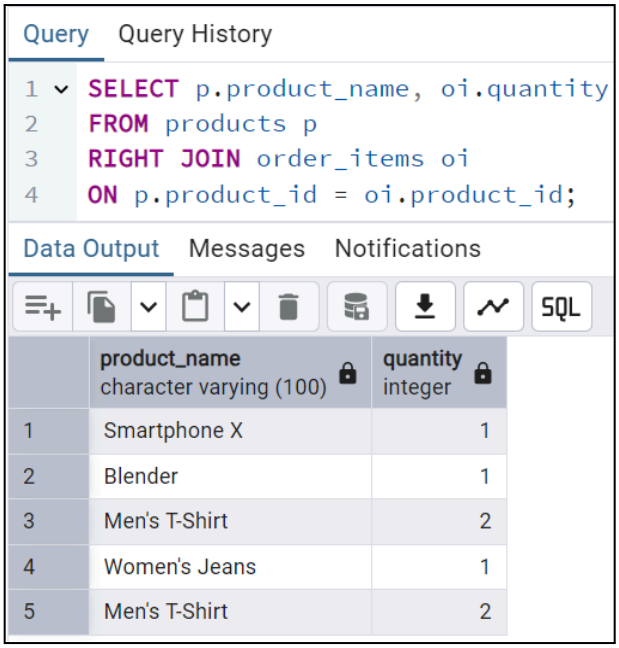
```
SELECT o.order_id, c.first_name, c.last_name, o.total_amount
FROM orders o INNER JOIN customers c
ON o.customer_id = c.customer_id;
```

	order_id integer	first_name character varying (50)	last_name character varying (50)	total_amount numeric (10,2)
1	1	John	Smith	799.99
2	3	Michael	Williams	59.99
3	4	John	Smith	79.97
4	5	John	Smith	39.98

```
SELECT c.customer_id, c.first_name, o.order_id
FROM customers c
LEFT JOIN orders o
ON c.customer_id = o.customer_id;
```

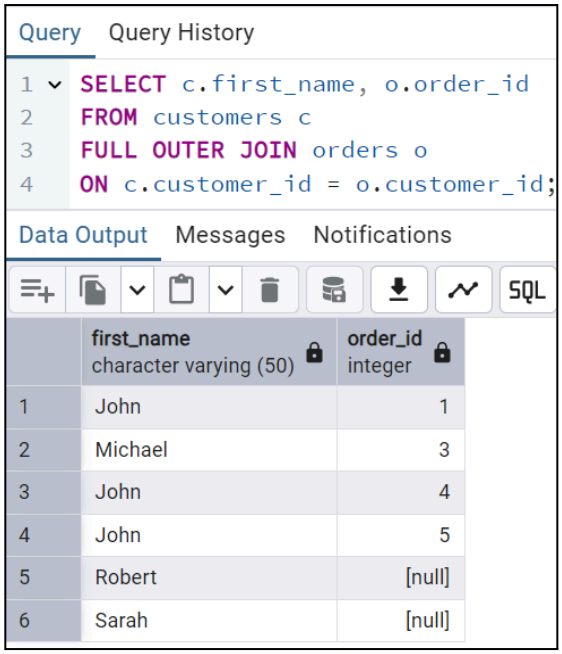
	customer_id integer	first_name character varying (50)	order_id integer
1	1	John	1
2	3	Michael	3
3	1	John	4
4	1	John	5
5	5	Robert	[null]
6	4	Sarah	[null]

LEFT JOIN: or LEFT OUTER JOIN returns all records from the left table and the matching records from the right table. If no match exists, the result will still include the row from the left table but fill the right side with NULLs. This join is useful when you want to retain all records from the primary (left) table, even if they don't have associated data in the right table.



```
SELECT p.product_name, oi.quantity
FROM products p
RIGHT JOIN order_items oi
ON p.product_id = oi.product_id;
```

	product_name character varying (100)	quantity integer
1	Smartphone X	1
2	Blender	1
3	Men's T-Shirt	2
4	Women's Jeans	1
5	Men's T-Shirt	2



```
SELECT c.first_name, o.order_id
FROM customers c
FULL OUTER JOIN orders o
ON c.customer_id = o.customer_id;
```

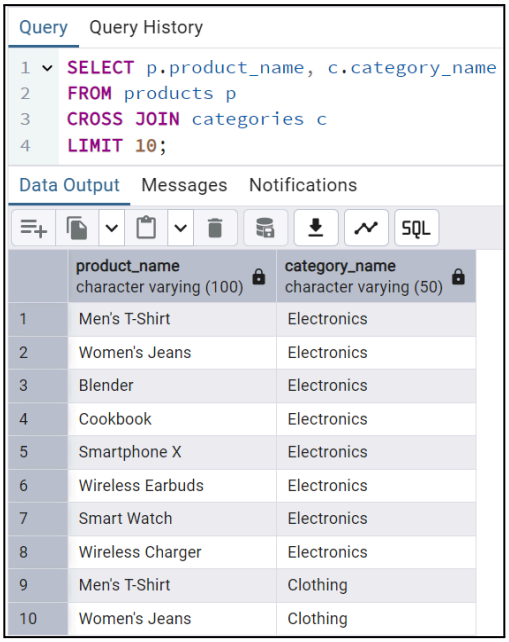
	first_name character varying (50)	order_id integer
1	John	1
2	Michael	3
3	John	4
4	John	5
5	Robert	[null]
6	Sarah	[null]

RIGHT JOIN: is the reverse of a LEFT JOIN. It returns all records from the right table and the matched records from the left table. Unmatched rows from the right table will appear with NULLs for the left table's columns. This join is helpful when you prioritize data from the right table and still want to see unmatched entries from it.

FULL OUTER JOIN: combines the results of both LEFT and RIGHT joins. It returns all rows from both tables, filling in NULLs where there is no match on either side. This type of join is ideal when you want a complete dataset from both sources, including unmatched records from either table.

CROSS JOIN: produces the Cartesian product of two tables, meaning it pairs every row from the first table with every row from the second. No join condition is used. While it can be useful in generating test data or creating combinations, it can result in very large datasets and should be used with caution.

SELF JOIN: is when a table is joined with itself. This is useful for comparing rows within the same table, such as identifying hierarchical relationships (e.g., employees and their managers) or finding duplicates. It requires aliasing the table to distinguish between the two instances in the query.

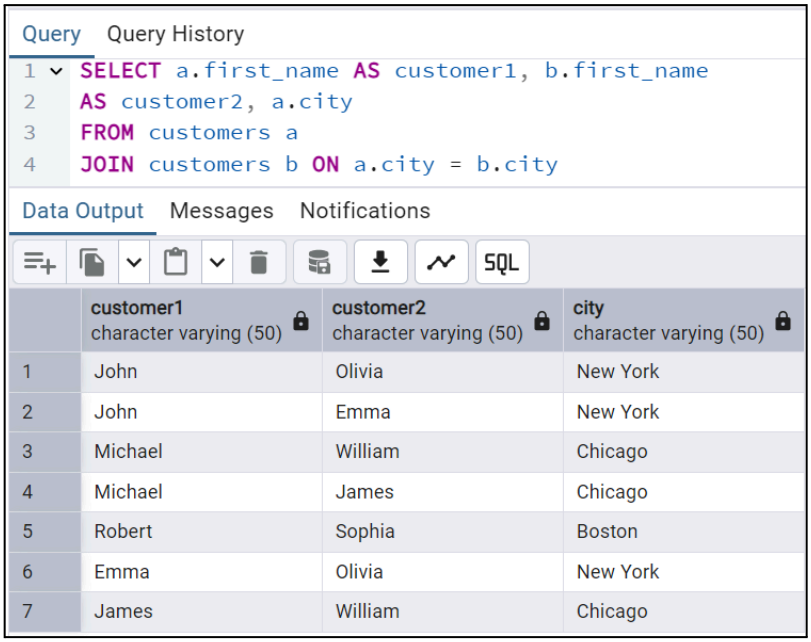


Query | Query History

```
SELECT p.product_name, c.category_name
FROM products p
CROSS JOIN categories c
LIMIT 10;
```

Data Output | Messages | Notifications

	product_name character varying (100)	category_name character varying (50)
1	Men's T-Shirt	Electronics
2	Women's Jeans	Electronics
3	Blender	Electronics
4	Cookbook	Electronics
5	Smartphone X	Electronics
6	Wireless Earbuds	Electronics
7	Smart Watch	Electronics
8	Wireless Charger	Electronics
9	Men's T-Shirt	Clothing
10	Women's Jeans	Clothing



Query | Query History

```
SELECT a.first_name AS customer1, b.first_name
AS customer2, a.city
FROM customers a
JOIN customers b ON a.city = b.city
```

Data Output | Messages | Notifications

	customer1 character varying (50)	customer2 character varying (50)	city character varying (50)
1	John	Olivia	New York
2	John	Emma	New York
3	Michael	William	Chicago
4	Michael	James	Chicago
5	Robert	Sophia	Boston
6	Emma	Olivia	New York
7	James	William	Chicago

Multi-table Joins: involve joining more than two tables in a single query. This is common in normalized databases where related data is spread across several tables. Each join condition connects one table to another, forming a chain. Proper join logic is essential to ensure accuracy and performance.

Query | Query History

```
SELECT o.order_id, c.first_name, p.product_name, oi.quantity
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
JOIN order_items oi ON o.order_id = oi.order_id
JOIN products p ON oi.product_id = p.product_id;
```

Data Output | Messages | Notifications

	order_id integer	first_name character varying (50)	product_name character varying (100)	quantity integer
1	1	John	Smartphone X	1
2	3	Michael	Blender	1
3	4	John	Men's T-Shirt	2
4	4	John	Women's Jeans	1
5	5	John	Men's T-Shirt	2

Joins with Filtering: apply conditions (using WHERE or ON) to refine the results of a join. This can limit the number of rows returned, focus on specific criteria (e.g., date range, status), and ensure data relevance. Filters can be applied before or after the join, depending on whether the condition is join-specific or general.

Query | Query History

```
SELECT p.product_name, p.price
FROM products p
JOIN categories c ON p.category_id = c.category_id
WHERE c.category_name = 'Electronics';
```

Data Output | Messages | Notifications

	product_name character varying (100)	price numeric (10,2)
1	Smartphone X	967.99
2	Wireless Earbuds	181.49
3	Smart Watch	241.99
4	Wireless Charger	36.29

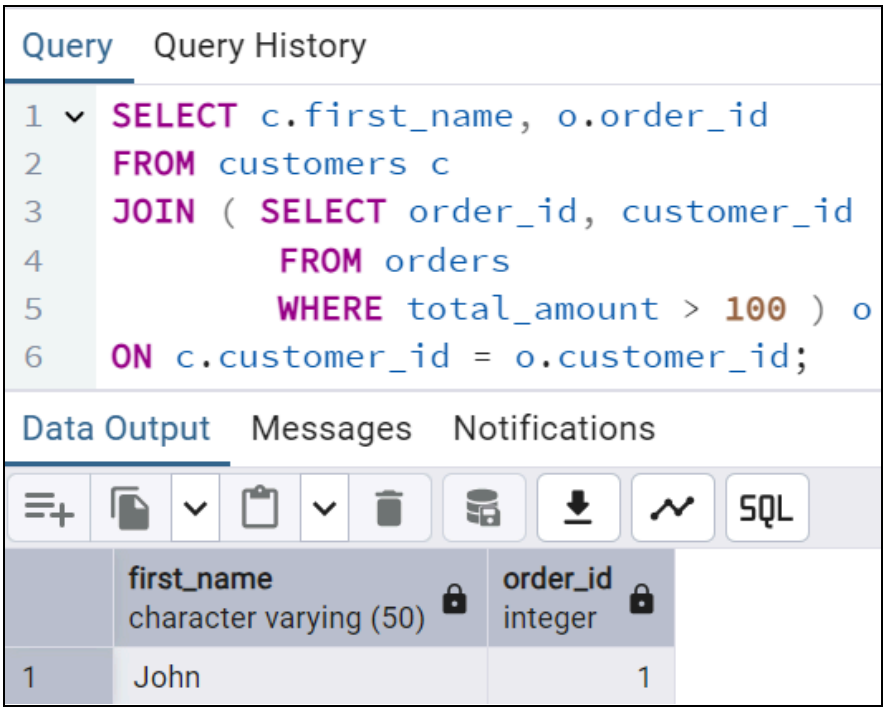
Query | Query History

```
SELECT c.category_name,
COUNT(p.product_id) AS product_count
FROM categories c
LEFT JOIN products p
ON c.category_id = p.category_id
GROUP BY c.category_name;
```

Data Output | Messages | Notifications

	category_name character varying (50)	product_count bigint
1	Home & Kitchen	1
2	Electronics	4
3	Clothing	2
4	Books	1

Joins with Aggregation: combine rows across tables and then apply aggregate functions such as SUM, COUNT, or AVG. This is common in reporting and analytics, for instance, joining orders with customers and calculating the total spend per customer. Grouping is typically done using GROUP BY after the join.



Relationship Mapping: refers to the way entities (tables) are linked via keys, such as primary and foreign keys. Joins rely on this mapping to connect data logically, for example, mapping orders to customers using customer_id. Correct mapping ensures accurate data retrieval across relational databases.

NULL Handling in Outer Joins: In outer joins, NULLs are used to represent missing or unmatched values from one side of the join. It's important to account for these NULLs in logic, especially in filters and calculations. Functions like COALESCE are commonly used to handle or replace NULLs to maintain data integrity and avoid misinterpretation.

Join Optimization: focuses on improving the performance of join queries. Techniques include indexing join keys, reducing dataset size using filters early, selecting only necessary columns, and choosing the appropriate type of join. Poorly optimized joins can lead to slow queries and unnecessary resource consumption, especially with large datasets.