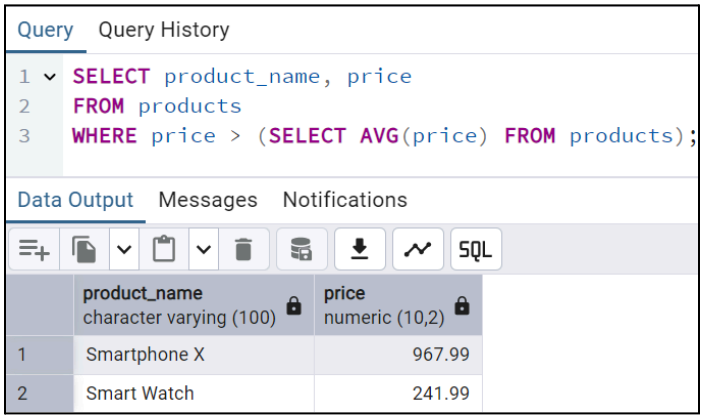
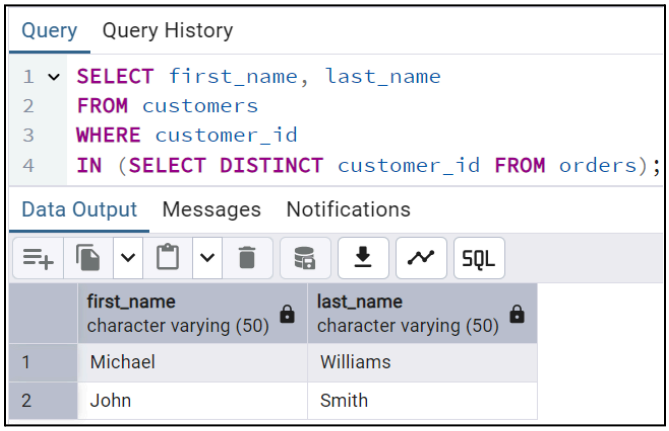
TASK-6

Scalar Subquery: is a subquery that returns exactly one value (a single row and a single column). It is often used in the SELECT list or in WHERE and HAVING clauses to compute a value based on a related query. For example, you might retrieve an employee's salary along with the average salary across all employees by using a scalar subquery. If the subquery returns more than one value, it will raise an error.



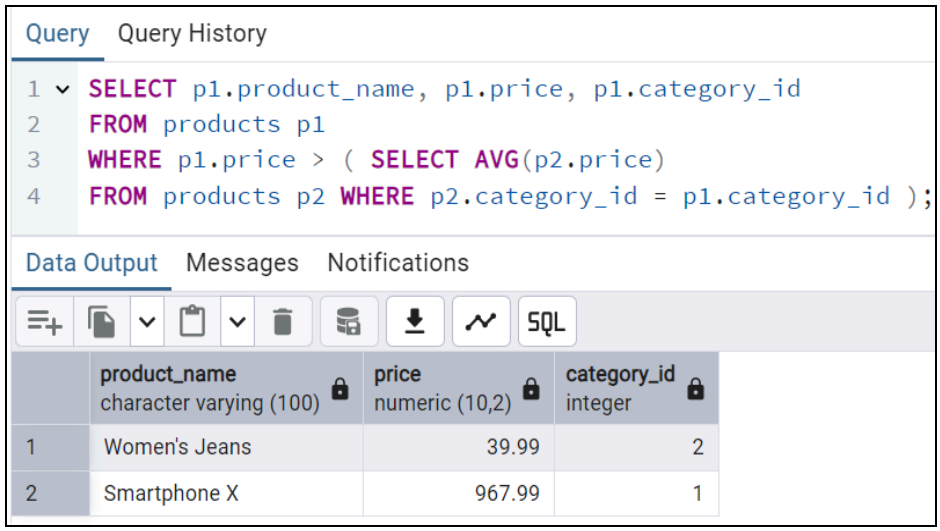
1. SCALAR SUBQUERY (Single value) Find products priced above average



2. MULTI-ROW SUBQUERY (IN operator) Customers who placed orders

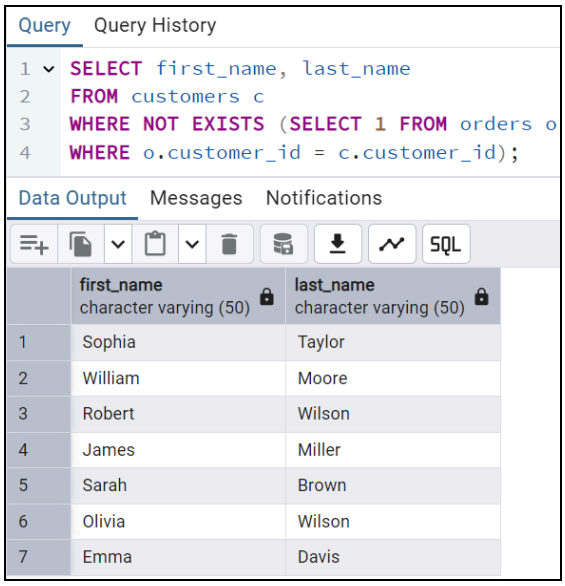
Multi-row Subquery: returns multiple rows and is typically used with operators like IN, ANY, or ALL. These are useful when you want to filter records based on a list of values produced by another query. For instance, selecting customers who have placed orders in a specific year by using a subquery that returns all customer IDs from that year's orders.

Correlated Subquery: depends on the outer query for its values. It is evaluated repeatedly, once for each row in the outer query. This makes it slower but powerful for row-wise comparisons, such as finding employees who earn more than the average salary in their respective departments. The inner query references columns from the outer query, creating a dynamic relationship.



3. CORRELATED SUBQUERY (Row-by-row) Products with above-average prices in their category

EXISTS / NOT EXISTS: The ***EXISTS*** clause checks for the existence of rows returned by a subquery. It returns TRUE if the subquery returns at least one row. ***NOT EXISTS*** does the opposite and returns TRUE when the subquery yields no rows. These are especially efficient with correlated subqueries for existence checks, such as checking whether a customer has placed any orders.

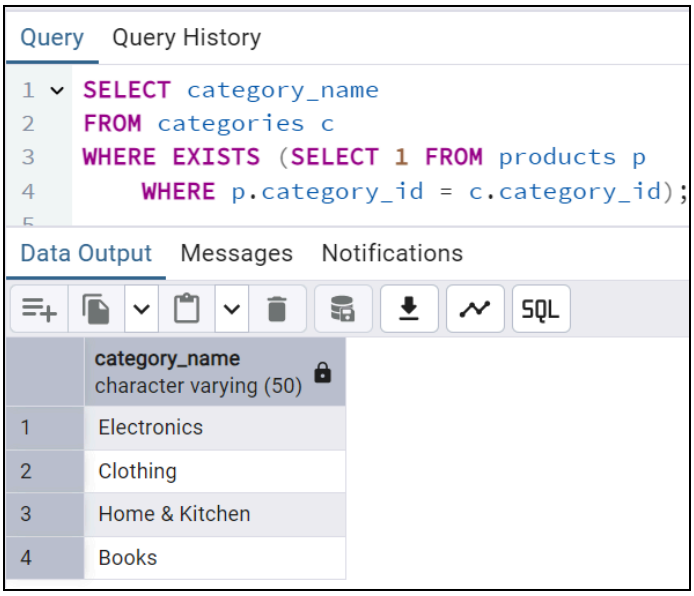


```
Query  Query History
1 SELECT first_name, last_name
2 FROM customers c
3 WHERE NOT EXISTS (SELECT 1 FROM orders o
4 WHERE o.customer_id = c.customer_id);
```

Data Output | Messages | Notifications

	first_name character varying (50)	last_name character varying (50)
1	Sophia	Taylor
2	William	Moore
3	Robert	Wilson
4	James	Miller
5	Sarah	Brown
6	Olivia	Wilson
7	Emma	Davis

4. NOT EXISTS SUBQUERY Customers with no orders



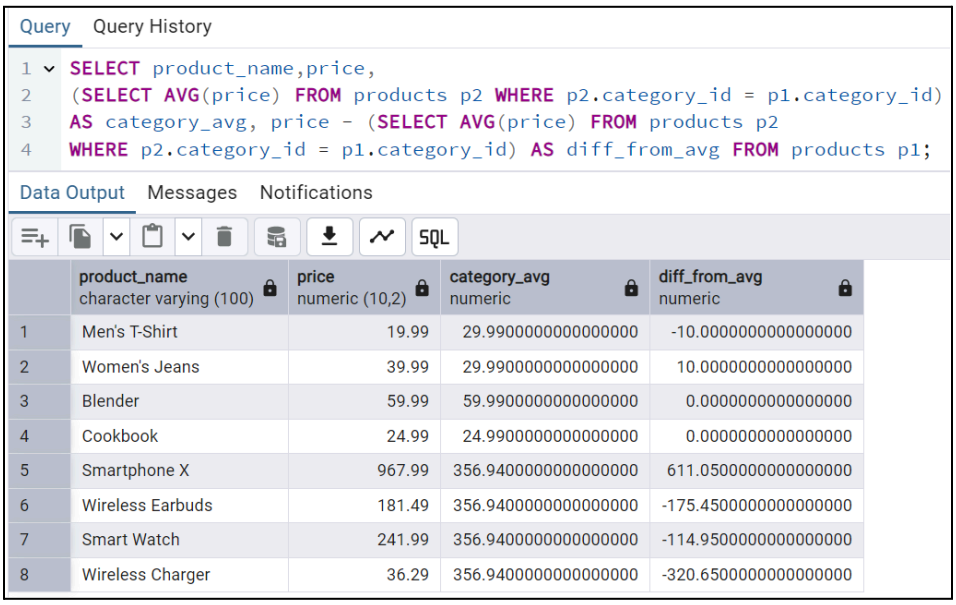
```
Query  Query History
1 SELECT category_name
2 FROM categories c
3 WHERE EXISTS (SELECT 1 FROM products p
4     WHERE p.category_id = c.category_id);
```

Data Output | Messages | Notifications

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

5. EXISTS SUBQUERY (Boolean check)Categories with products

Derived Tables (FROM Clause Subqueries): are subqueries placed inside the FROM clause, allowing you to treat their output as a temporary table for the rest of the query. They are useful for organizing complex logic, pre-aggregating data, or simplifying deeply nested operations. Aliasing is mandatory for derived tables so the outer query can refer to them.



```
Query  Query History
1 SELECT product_name,price,
2 (SELECT AVG(price) FROM products p2 WHERE p2.category_id = p1.category_id)
3 AS category_avg, price - (SELECT AVG(price) FROM products p2
4 WHERE p2.category_id = p1.category_id) AS diff_from_avg FROM products p1;
```

Data Output | Messages | Notifications

	product_name character varying (100)	price numeric (10,2)	category_avg numeric	diff_from_avg numeric
1	Men's T-Shirt	19.99	29.9900000000000000	-10.0000000000000000
2	Women's Jeans	39.99	29.9900000000000000	10.0000000000000000
3	Blender	59.99	59.9900000000000000	0.0000000000000000
4	Cookbook	24.99	24.9900000000000000	0.0000000000000000
5	Smartphone X	967.99	356.9400000000000000	611.0500000000000000
6	Wireless Earbuds	181.49	356.9400000000000000	-175.4500000000000000
7	Smart Watch	241.99	356.9400000000000000	-114.9500000000000000
8	Wireless Charger	36.29	356.9400000000000000	-320.6500000000000000

6. SUBQUERY IN SELECT clause Product list with price comparison to category avg

Query | Query History

```
1  SELECT c.first_name, c.last_name, avg_orders.avg_amount
2  FROM customers c JOIN (SELECT customer_id,
3  AVG(total_amount) AS avg_amount FROM orders GROUP BY customer_id)
4  avg_orders ON c.customer_id = avg_orders.customer_id;
```

Data Output | Messages | Notifications

	first_name character varying (50)	last_name character varying (50)	avg_amount numeric
1	Michael	Williams	59.9900000000000000
2	John	Smith	306.6466666666666667

7. DERIVED TABLE (Subquery in FROM) Average order value by customer

Query | Query History

```
1  SELECT o.order_id, c.first_name, o.total_amount FROM orders o
2  JOIN customers c ON o.customer_id = c.customer_id
3  WHERE c.customer_id IN (SELECT customer_id FROM orders
4  GROUP BY customer_id HAVING SUM(total_amount) > 300 );
```

Data Output | Messages | Notifications

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

8. SUBQUERY WITH JOIN Order details for high-value customers

LATERAL Joins : allow subqueries in the FROM clause to reference columns from preceding tables. This is especially useful when applying a function or subquery to each row of another table, such as selecting the top N items per category. LATERAL is a PostgreSQL feature that extends SQL's flexibility in query design and supports more dynamic data processing.

Query | Query History

```
1  SELECT c.category_name, p.product_name, p.price
2  FROM categories c, LATERAL (SELECT product_name,
3  price FROM products WHERE category_id = c.category_id
4  ORDER BY price DESC LIMIT 1) p;
```

Data Output | Messages | Notifications

	category_name character varying (50)	product_name character varying (100)	price numeric (10,2)
1	Electronics	Smartphone X	967.99
2	Clothing	Women's Jeans	39.99
3	Home & Kitchen	Blender	59.99
4	Books	Cookbook	24.99

9. LATERAL SUBQUERY Top product from each category

Recursive Common Table Expressions: allow you to perform recursive queries, ideal for hierarchical or graph-like data structures, such as organization trees or folder paths. The CTE consists of a base query (anchor) and a recursive query that references the CTE itself. PostgreSQL supports recursion through the WITH RECURSIVE keyword, enabling elegant traversal of depth-based structures.

Query | Query History

```
1  WITH RECURSIVE category_tree AS
2  (SELECT category_id, category_name, 1 AS level
3  FROM categories WHERE category_id = 1
4  UNION ALL SELECT c.category_id,
5  c.category_name, ct.level + 1
6  FROM categories c JOIN category_tree ct
7  ON c.category_id = ct.category_id + 1 )
8  SELECT * FROM category_tree;
```

Data Output | Messages | Notifications

	category_id integer	category_name character varying (50)	level integer
1	1	Electronics	1
2	2	Clothing	2
3	3	Home & Kitchen	3
4	4	Books	4

10. LATERAL SUBQUERY Top product from each category