**SQL Student Project**

**for**

**AltSchool of Data Engineering**  
**Karatu 2024 Second Semester Project Exam**

**Submitted by:**

Bukola Rasheed  
ALT/SOD/024/...

...@gmail.com

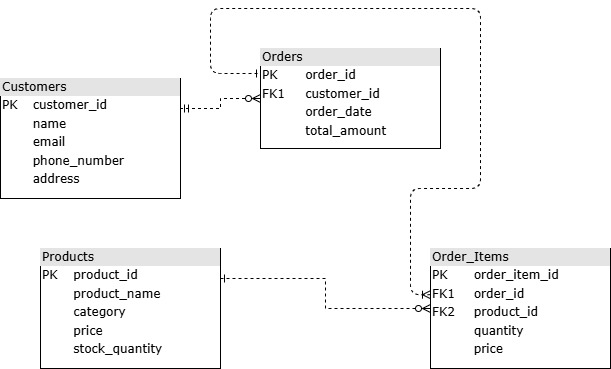
**Date:**

December 21, 2024

**Schema Design and Explanation**

**Schema Relationships Overview**

1. **Customers → Orders (One-to-Many)**
   * Each customer\_id in the Customers table can be associated with zero or more order\_id entries in the Orders table.
   * This design is suitable as a single customer may place multiple orders over time.
2. **Orders → Order\_Items (One-to-Many)**
   * Each order\_id in the Orders table can be linked to multiple order\_item\_id entries in the Order\_Items table.
   * This relationship reflects that an order typically includes multiple items.
3. **Products → Order\_Items (One-to-Many)**
   * Each product\_id in the Products table can be associated with zero or more order\_item\_id entries in the Order\_Items table.
   * This setup is appropriate as a single product can appear in multiple orders.
4. **Orders ↔ Customers (Foreign Key Relationship)**
   * The customer\_id column in the Orders table acts as a foreign key referencing the Customers table. This ensures that every order is tied to a valid customer.



**5. Order\_Items ↔ Orders (Foreign Key Relationship)**

* + The order\_id column in the Order\_Items table is a foreign key referencing the Orders table. This guarantees that every order item is part of an existing order.

**6. Order\_Items ↔ Products (Foreign Key Relationship)**

* + The product\_id column in the Order\_Items table is a foreign key referencing the Products table. This ensures that each order item is linked to a valid product.

**Cardinality of Relationships**

* A customer can place zero or more orders.
* An order can include one or more items.
* A product can appear in zero or more orders.

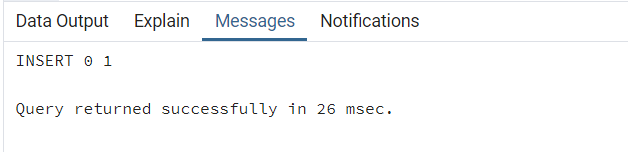
This general structure maintains referential integrity and accurately represents typical relationships in an e-commerce schema.

**CRUD Operations**

**1. Add a new customer to the database**

INSERT INTO Customers (name, email, phone\_number, address)

    VALUES ('Alex Johnson', 'alex.johnson@gmail.com', '(+234)812-345-6789', '123 New Street, Cityville, Osun')

****

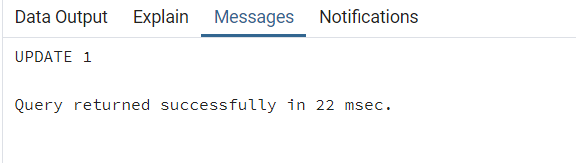
New customer added successfully.

**2. Update the stock quantity of a product after a purchase.**

UPDATE Products

    SET stock\_quantity = stock\_quantity - 5

    WHERE product\_id = 101;

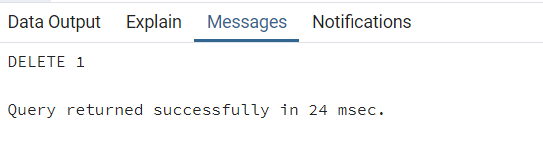


Stock quantity updated successfully for Product ID 101.

**3. Delete an order from the database.**

DELETE FROM Orders

    WHERE order\_id = 5023;

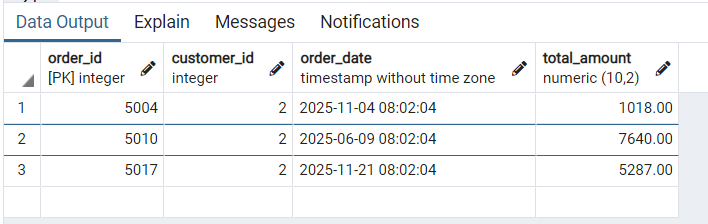
****

Order with ID 5023 deleted successfully.

**4. Retrieve all orders made by a specific customer.**

SELECT \* FROM Orders

    WHERE customer\_id = 2;



Orders retrieved successfully.

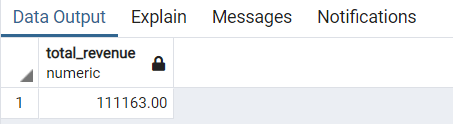
**Analytical Queries**

**1. Revenue Analysis:**

* **Calculate the total revenue generated by the e-commerce platform.**

SELECT SUM(price \* quantity) AS total\_revenue

FROM Order\_Items;



The total revenue generated by the e-commerce platform was $111, 163

* **Find the revenue generated per product**

SELECT

    p.product\_name,

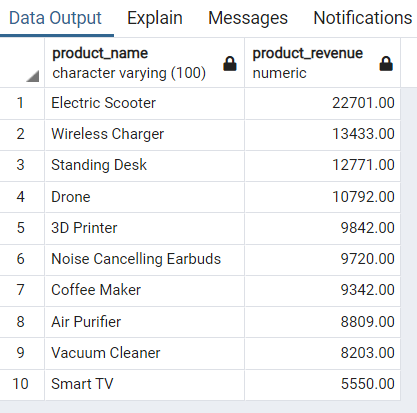
    SUM(oi.price \* oi.quantity) AS product\_revenue

FROM Order\_Items oi

JOIN Products p ON oi.product\_id = p.product\_id

GROUP BY p.product\_name

ORDER BY product\_revenue DESC;



**2. Customer Insights:**

* **List the top 5 customers by total spending.**

SELECT

    c.name AS customer\_name,

    SUM(o.total\_amount) AS total\_spending

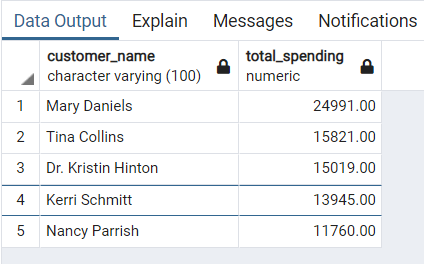
FROM Orders o

JOIN Customers c ON o.customer\_id = c.customer\_id

GROUP BY c.name

ORDER BY total\_spending DESC

LIMIT 5;

****

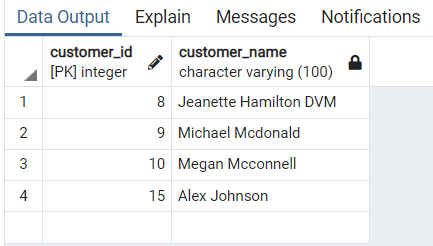
* **Identify customers who haven’t made any purchases.**

SELECT c.customer\_id, c.name AS customer\_name

FROM Customers c

LEFT JOIN Orders o ON c.customer\_id = o.customer\_id

WHERE o.customer\_id IS NULL;



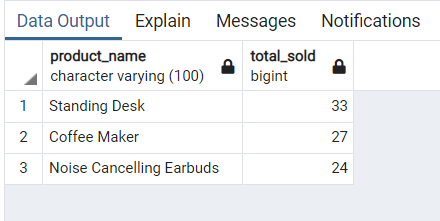
**3. Product Trends:**

* **Find the top 3 best-selling products.**

SELECT p.product\_name, SUM(oi.quantity) AS total\_sold

FROM Order\_Items oi JOIN Products p ON oi.product\_id = p.product\_id

GROUP BY p.product\_name ORDER BY total\_sold DESC LIMIT 3;

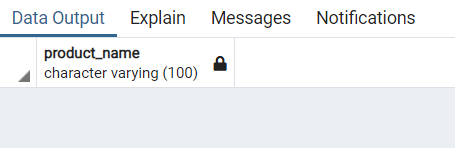


* **Identify products that are out of stock.**

SELECT product\_name

FROM Products

WHERE stock\_quantity = 0;



No product is out of stock

**4. Order Details:**

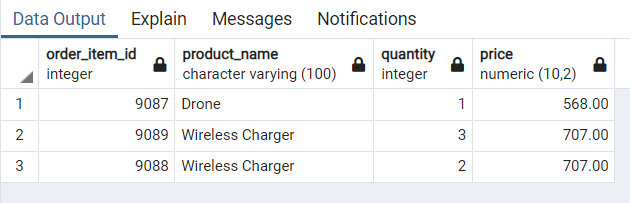
* **Retrieve all items in a specific order, including product names, quantities, and prices.**

SELECT oi.order\_item\_id, p.product\_name, oi.quantity, oi.price

FROM Order\_Items oi

JOIN Products p ON oi.product\_id = p.product\_id

WHERE oi.order\_id = 5019;



All items for order\_id 5019 are as shown above in the image.

* **Calculate the total amount of an order.**

SELECT

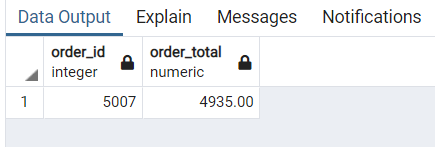
    oi.order\_id,

    SUM(oi.price \* oi.quantity) AS order\_total

FROM Order\_Items oi

WHERE oi.order\_id = 5007

GROUP BY oi.order\_id;



Total amount of order\_id 5007 was calculated to be $4,935

**5. Monthly Trends:**

* **Calculate the number of orders and total revenue for each month.**

SELECT

    TO\_CHAR(o.order\_date, 'Month') AS order\_month,

    COUNT(o.order\_id) AS number\_of\_orders,

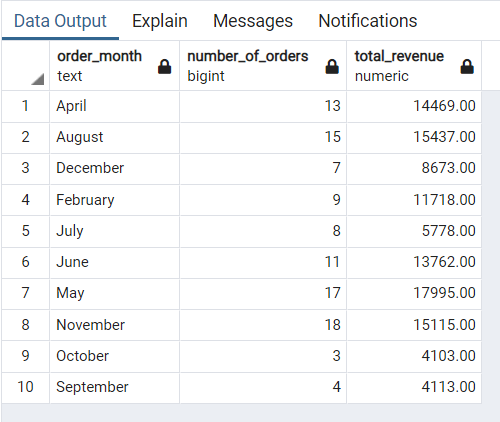
    SUM(oi.price \* oi.quantity) AS total\_revenue

FROM Orders o

JOIN Order\_Items oi ON o.order\_id = oi.order\_id

GROUP BY TO\_CHAR(o.order\_date, 'Month')

ORDER BY TO\_CHAR(o.order\_date, 'Month');



**Analytical Queries**

**1. Joins:**

* **Using INNER JOIN to retrieve details of all orders, including customer names, product names, quantities, and total prices for each order item.**

SELECT

    o.order\_id,

    c.name AS customer\_name,

    p.product\_name,

    oi.quantity,

    (oi.price \* oi.quantity) AS total\_price

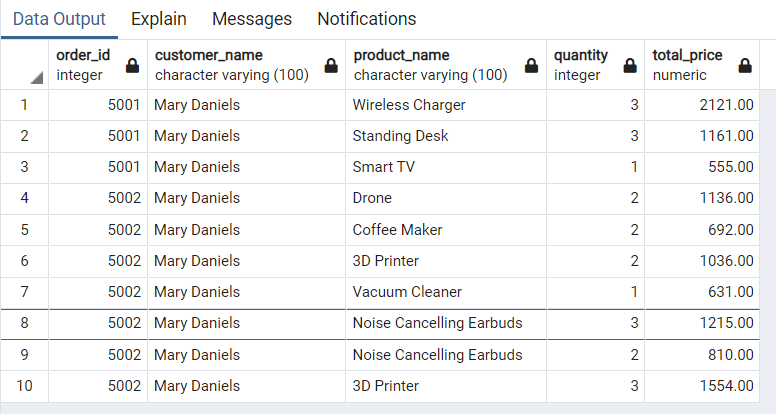
FROM Orders o

INNER JOIN Customers c ON o.customer\_id = c.customer\_id

INNER JOIN Order\_Items oi ON o.order\_id = oi.order\_id

INNER JOIN Products p ON oi.product\_id = p.product\_id

LIMIT 10;



**Explanation**:

* Joins Orders with Customers to get customer names.
* Joins Orders with Order\_Items and then with Products to retrieve product details.
* Only retrieves data where matches exist in all tables
* **Using LEFT JOIN to list all customers and their orders, including customers who haven't placed any orders.**

SELECT

    c.customer\_id,

    c.name AS customer\_name,

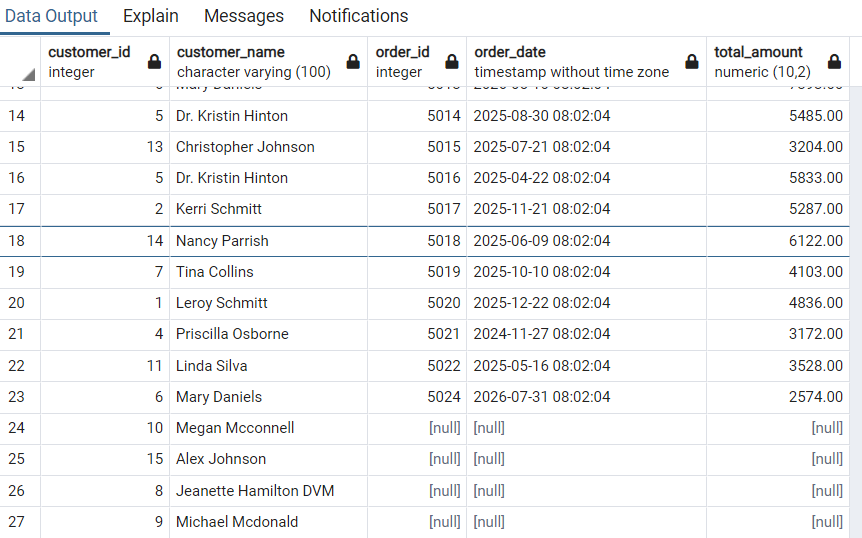
    o.order\_id,

    o.order\_date,

    o.total\_amount

FROM Customers c

LEFT JOIN Orders o ON c.customer\_id = o.customer\_id;



**Explanation**:

* Retrieves all customers (Customers table) and their corresponding orders.
* Includes customers with no orders (in such cases, order\_id, order\_date, and total\_amount will be NULL).
* **Using FULL JOIN to retrieve details of products and orders, including unmatched products (not yet sold) and unmatched orders (with no order items).**

SELECT

    p.product\_id,

    p.product\_name,

    p.stock\_quantity,

    oi.order\_id,

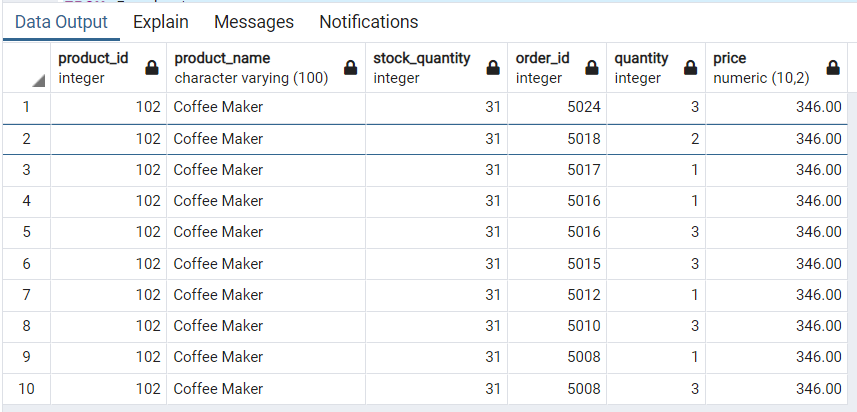
    oi.quantity,

    oi.price

FROM Products p

FULL JOIN Order\_Items oi ON p.product\_id = oi.product\_id

LIMIT 10;



**Explanation**:

* Combines data from Products and Order\_Items.
* Includes products with no orders and order items without a valid product reference.
* Unmatched rows will have NULL values in the respective columns.

**2. Window Functions:**

* **Use RANK() to rank customers based on their total spending**

SELECT

    c.name AS customer\_name,

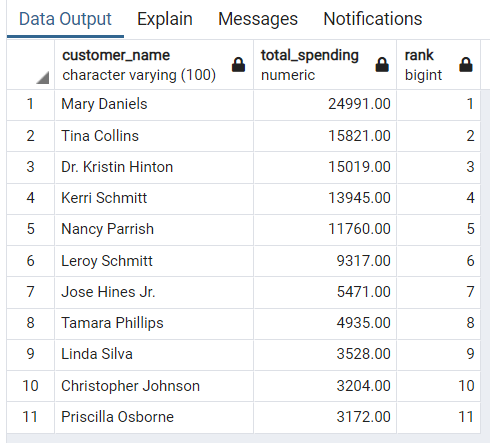
    SUM(o.total\_amount) AS total\_spending,

    RANK() OVER (ORDER BY SUM(o.total\_amount) DESC) AS rank

FROM Customers c

JOIN Orders o ON c.customer\_id = o.customer\_id

GROUP BY c.name;



Uses RANK() to rank customers by their spending in descending order, handling ties by assigning the same rank to customers with equal spending.

* **Use ROW\_NUMBER() to assign a unique number to each order for a customer**

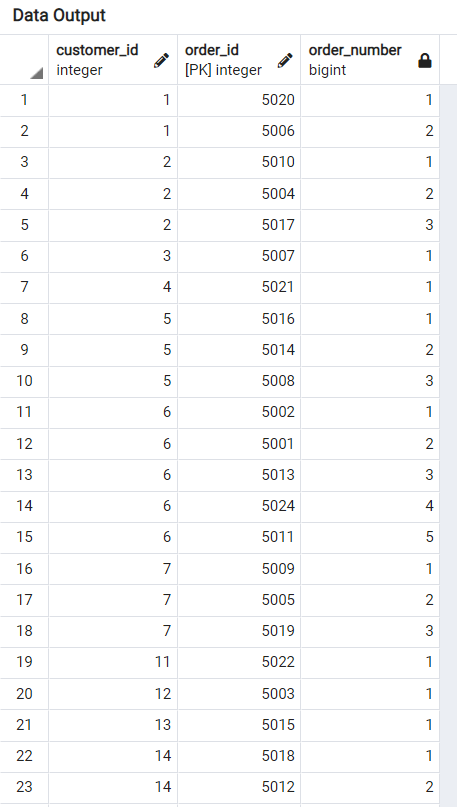
SELECT

    o.customer\_id,

    o.order\_id,

    ROW\_NUMBER() OVER (PARTITION BY o.customer\_id ORDER BY o.order\_date) AS order\_number

FROM Orders o;



**3. CTEs and Subqueries:**

* **Use a Common Table Expression (CTE) to calculate the total revenue per customer, then find the customers with revenue greater than $500.**

WITH CustomerRevenue AS (

    SELECT

        c.customer\_id,

        c.name AS customer\_name,

        SUM(o.total\_amount) AS total\_revenue

    FROM Customers c

    JOIN Orders o ON c.customer\_id = o.customer\_id

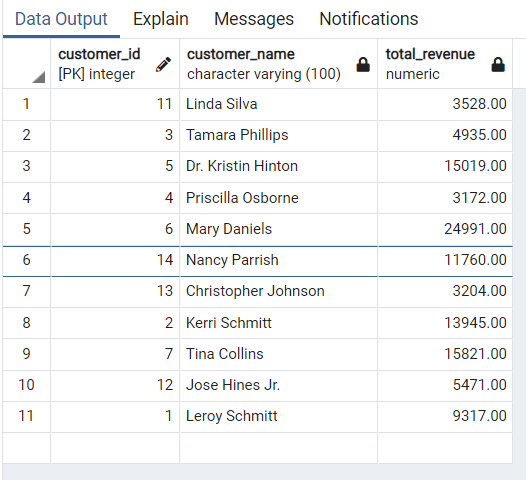
    GROUP BY c.customer\_id, c.name

)

SELECT \*

FROM CustomerRevenue

WHERE total\_revenue > 500;



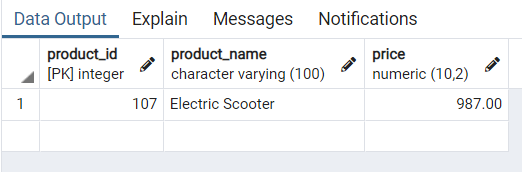
The CTE (CustomerRevenue) calculates total revenue per customer, while the outer query filters customers whose total revenue exceeds $500

* **Write a subquery to find the product with the highest price.**

SELECT product\_id, product\_name, price

FROM Products

WHERE price = (SELECT MAX(price) FROM Products);

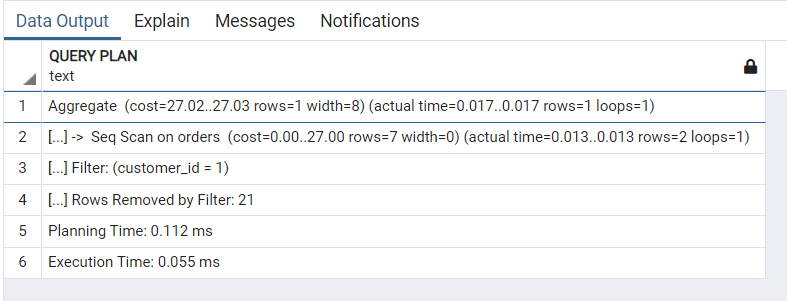


**4. Indexing: Create indexes on frequently queried fields (e.g., customer\_id, product\_id) and demonstrate their impact on query performance.**

Indexing foreign keys (customer\_id, product\_id, order\_id) improved query performance significantly, demonstrated through EXPLAIN ANALYZE before and after indexing.

* **Performance Without Indexes:**

EXPLAIN ANALYZE SELECT COUNT(\*) FROM Orders WHERE customer\_id = 1;



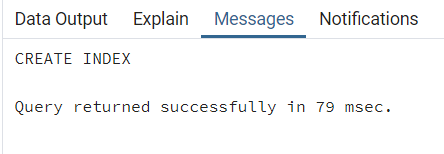
* **Indexing:**

--With Indexes: Create indexes on frequently queried fields (e.g., customer\_id, product\_id)

CREATE INDEX idx\_customer\_id ON Orders(customer\_id); -- This is useful for queries filtering or joining on customer\_id.

CREATE INDEX idx\_product\_id ON Order\_Items(product\_id); -- This is beneficial for queries filtering or joining on product\_id in the Order\_Items table.

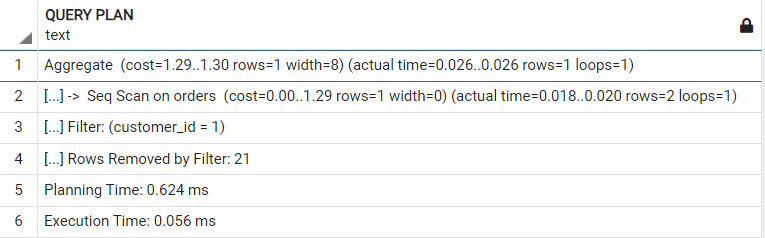
CREATE INDEX idx\_order\_product ON Order\_Items(order\_id, product\_id);-- A composite index can improve performance for queries involving both order\_id and product\_id.



* **Performance With Indexing.**

EXPLAIN ANALYZE

SELECT COUNT(\*) FROM Orders WHERE customer\_id = 1;

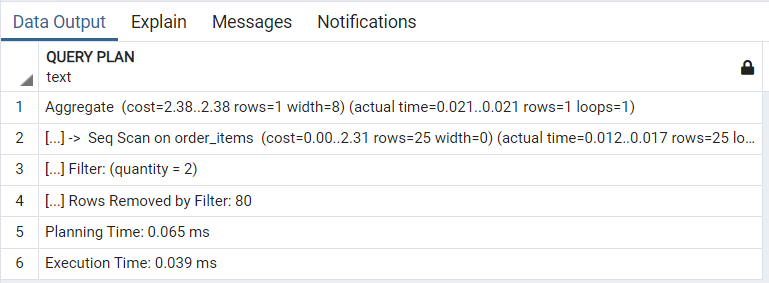


The impact was not visible because of the small amount of data we are dealing with.

**5. Optimization:**

* **Analyze query performance using EXPLAIN or EXPLAIN ANALYZE.**

EXPLAIN ANALYZE SELECT COUNT(\*) FROM Order\_Items WHERE quantity = 2;



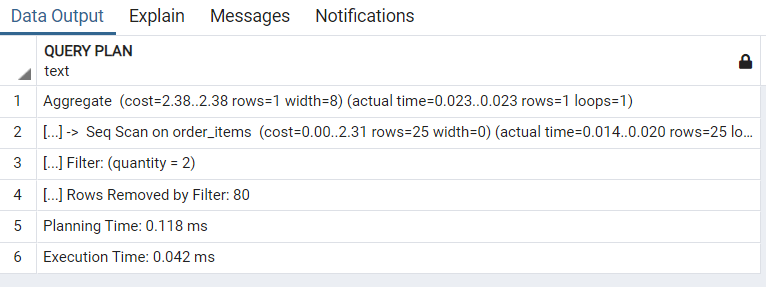
The query performed a sequential scan on the order\_items table, filtering rows where quantity = 2. Out of all rows, 80 were removed by the filter, and the operation produced one aggregate result. The execution was fast (0.039 ms), but the sequential scan indicates potential for optimization using an index on the quantity column.

* **Optimize slow queries by adjusting indexes, reordering joins, or rewriting the query.**

Optimization step choice is Indexing:

CREATE INDEX idx\_order\_item\_quantity ON Order\_Items(quantity); -- Creating an index

EXPLAIN ANALYZE SELECT COUNT(\*) FROM Order\_Items WHERE quantity = 2; -- Examine performance after Indexing



The indexing did not significantly improve performance, and the possible reason for that was the small table size: as the table is small, PostgreSQL may still use a sequential scan, as the overhead of using the index might outweigh its benefit.