AltSchool of Data Engineering

Karatu 2024 Second Semester Project Exam

Project Overview: E-Commerce Database Management and Analysis

This project will test your SQL knowledge, from database design and CRUD operations to advanced concepts like joins, aggregation, indexing, and optimization. The project involves designing a relational database for an e-commerce platform, performing data manipulations, and extracting meaningful insights using SQL queries.

Your task is to design, implement, and query an SQL database for a fictional e-commerce platform. The database will include tables for customers, products, orders, and order_items. You will perform data operations, write queries to answer analytical questions, and demonstrate optimization techniques.

Submitted by:

Name: Gideon A. Ayanwoye

AltSchool ID: ALT/SOD/024/0995

Email: ayanwoyegideon@gmail.com

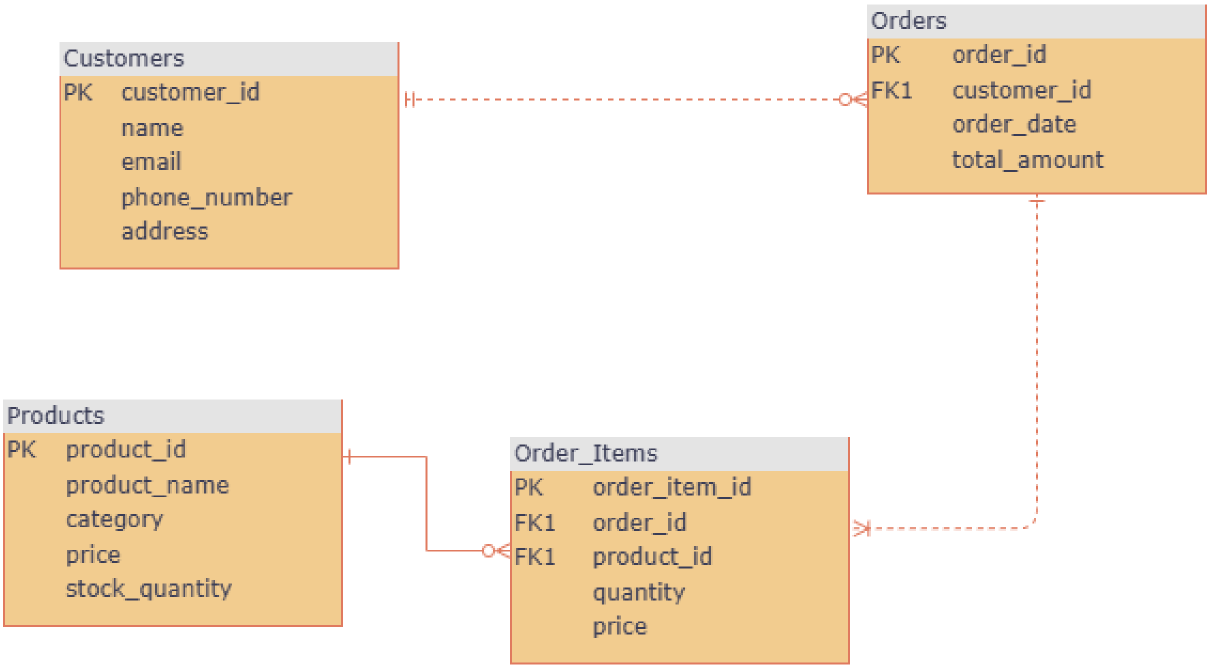
Date:

December 19, 2024

Course:

SQL Student Project (Data Engineering)

Schema Design and Explanation



Below is a breakdown of the relationships specified in the schema:

1. **Customers → Orders (One-to-Many)**
   - A ***customer_id*** in the **Customers** table can have zero or more corresponding ***order_id*** entries in the **Orders** table.
   - This relationship is appropriate since a single customer can place multiple orders.

2. **Orders → Order_Items (One-to-Many)**
   - An ***order_id*** in the **Orders** table can have multiple corresponding ***order_item_id*** entries in the **Order_Items** table.
   - This relationship is logical because each order can consist of multiple items.

3. **Products → Order_Items (One-to-Many)**
   - A ***product_id*** in the **Products** table can have zero or more corresponding ***order_item_id*** entries in the **Order_Items** table.
   - This is also correct since a single product can be part of multiple orders.

4. **Orders ↔ Customers (Foreign Key Relationship)**
   - The ***customer_id*** in the **Orders** table is a foreign key referencing the **Customers** table. This ensures that each order is linked to a valid customer.

5. **Order_Items ↔ Orders (Foreign Key Relationship)**
   - The ***order_id*** in the **Order_Items** table is a foreign key referencing the **Orders** table. This ensures each order item belongs to a valid order.

6. **Order_Items ↔ Products (Foreign Key Relationship)**

   - The ***product_id*** in the **Order_Items** table is a foreign key referencing the **Products** table. This ensures each order item is linked to a valid product.

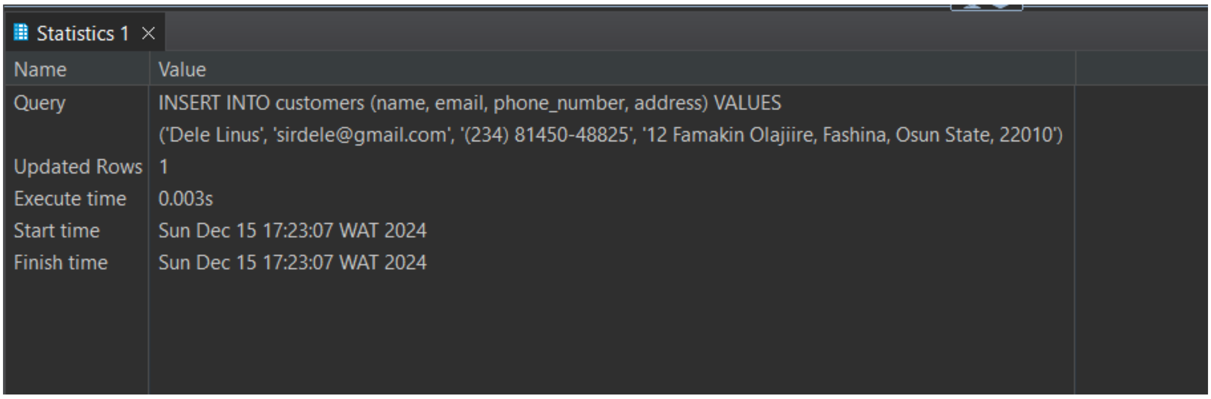
The **cardinality** of each relationship:

- A customer can have zero or more orders.
- An order can have one or more items.
- A product can be part of zero or more orders.

CRUD Operations

1. Add a new customer to the database

```
 INSERT INTO customers (name, email, phone_number, address) VALUES
('Dele Linus', 'sirdele@gmail.com', '(234) 81450-48825', '12 Famakin Olajiire,
Fashina, Osun State, 22010');
```

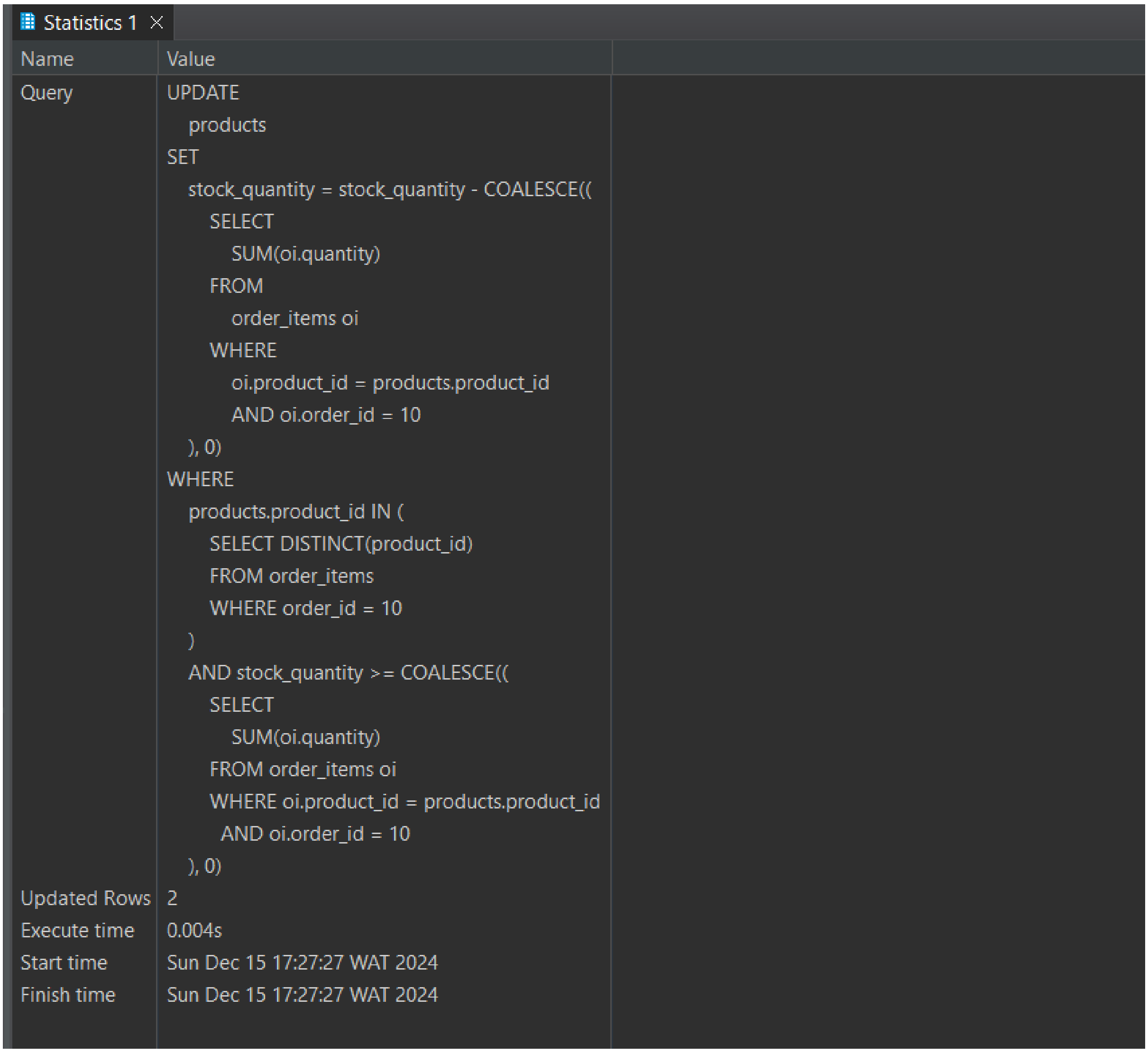


Name	Value
Query	INSERT INTO customers (name, email, phone_number, address) VALUES ('Dele Linus', 'sirdele@gmail.com', '(234) 81450-48825', '12 Famakin Olajiire, Fashina, Osun State, 22010')
Updated Rows	1
Execute time	0.003s
Start time	Sun Dec 15 17:23:07 WAT 2024
Finish time	Sun Dec 15 17:23:07 WAT 2024

Figure 2: Adding a new customer to the database

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

```
UPDATE
    products
SET
    stock_quantity = stock_quantity - COALESCE((
        SELECT
            SUM(oi.quantity)
        FROM
            order_items oi
        WHERE
            oi.product_id = products.product_id
            AND oi.order_id = 10
    ), 0)
WHERE
    products.product_id IN (
        SELECT DISTINCT(product_id)
        FROM order_items
        WHERE order_id = 10
    )
    AND stock_quantity >= COALESCE((
        SELECT
            SUM(oi.quantity)
        FROM order_items oi
        WHERE oi.product_id = products.product_id
          AND oi.order_id = 10
    ), 0);
```

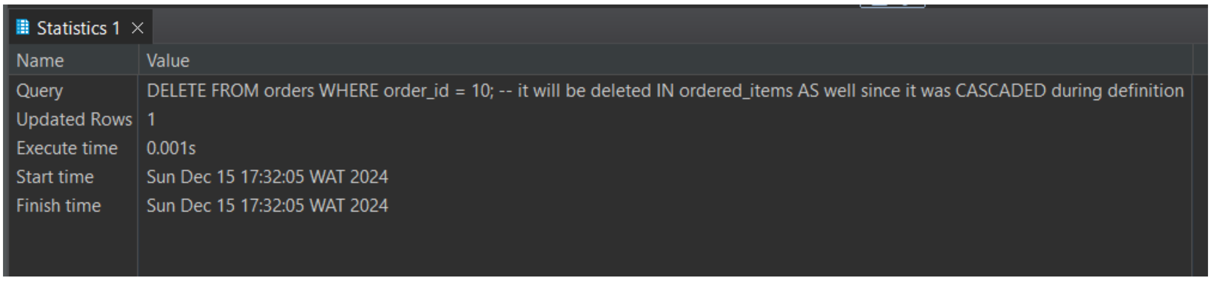


```
Name           Value
Query          UPDATE
                 products
               SET
                 stock_quantity = stock_quantity - COALESCE((
                   SELECT
                     SUM(oi.quantity)
                   FROM
                     order_items oi
                   WHERE
                     oi.product_id = products.product_id
                     AND oi.order_id = 10
                 ), 0)
               WHERE
                 products.product_id IN (
                   SELECT DISTINCT(product_id)
                   FROM order_items
                   WHERE order_id = 10
                 )
                 AND stock_quantity >= COALESCE((
                   SELECT
                     SUM(oi.quantity)
                   FROM order_items oi
                   WHERE oi.product_id = products.product_id
                    AND oi.order_id = 10
                 ), 0)
Updated Rows   2
Execute time   0.004s
Start time     Sun Dec 15 17:27:27 WAT 2024
Finish time    Sun Dec 15 17:27:27 WAT 2024
```

Figure 3: Updating the stock quantity of a product after a purchase

3. Delete an order from the database.

```
DELETE FROM orders WHERE order_id = 10; -- it will be deleted IN ordered_items AS
well since it was CASCADED during definition
```

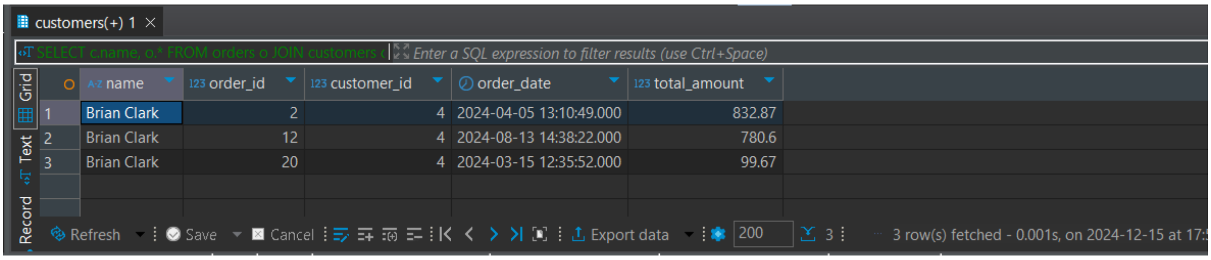


```
Name           Value
Query          DELETE FROM orders WHERE order_id = 10; -- it will be deleted IN ordered_items AS well since it was CASCADED during definition
Updated Rows   1
Execute time   0.001s
Start time     Sun Dec 15 17:32:05 WAT 2024
Finish time    Sun Dec 15 17:32:05 WAT 2024
```

Figure 4: Deleting order with id 10 from the database

4. Retrieve all orders made by a specific customer.

```
SELECT c.name, o.* FROM orders o JOIN customers c ON o.customer_id = c.customer_id
WHERE c.name = 'Brian Clark';
```



customers(+) 1
SELECT c.name, o.* FROM orders o JOIN customers c
Enter a SQL expression to filter results (use Ctrl+Space)
Grid
Text
Record
Refresh
Save
Cancel
Export data
200
3

	name	order_id	customer_id	order_date	total_amount
1	Brian Clark	2	4	2024-04-05 13:10:49.000	832.87
2	Brian Clark	12	4	2024-08-13 14:38:22.000	780.6
3	Brian Clark	20	4	2024-03-15 12:35:52.000	99.67

3 row(s) fetched - 0.001s, on 2024-12-15 at 17:

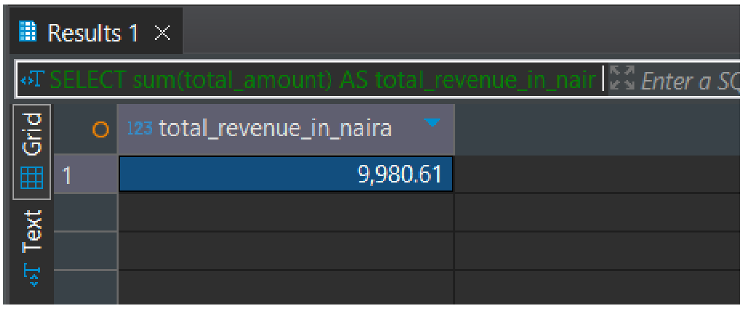
Figure 5: Retrieving all orders made by Brian Clark

Analytical Queries

1. Revenue Analysis:

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

```
SELECT sum(total_amount) AS total_revenue_in_naira FROM orders;
```

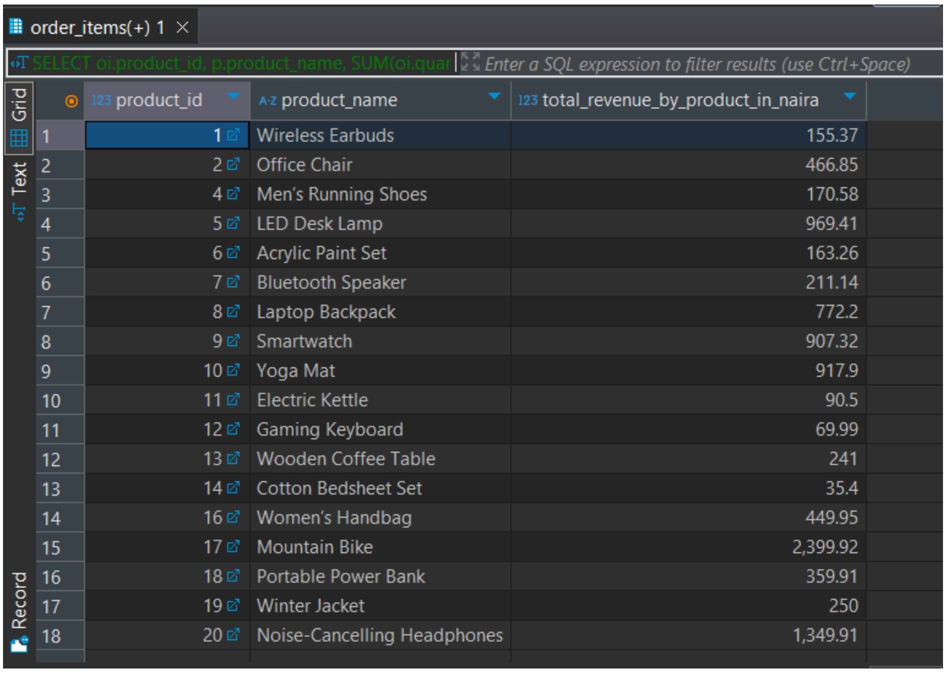


	total_revenue_in_naira
1	9,980.61

The total revenue generated by the platform was calculated using the **SUM(total_amount)** query from the **orders** table and it was 9,980.61 Naira.

- **Find the revenue generated per product**

```
SELECT
	oi.product_id, p.product_name,
	SUM(oi.quantity * oi.price) AS total_revenue_by_product_in_naira
FROM
	orders o
JOIN order_items oi ON (o.order_id = oi.order_id)
JOIN products p ON (p.product_id = oi.product_id)
GROUP BY oi.product_id, p.product_name ORDER BY oi.product_id;
```



SELECT oi.product_id, p.product_name, SUM(oi.quar | Enter a SQL expression to filter results (use Ctrl+Space)

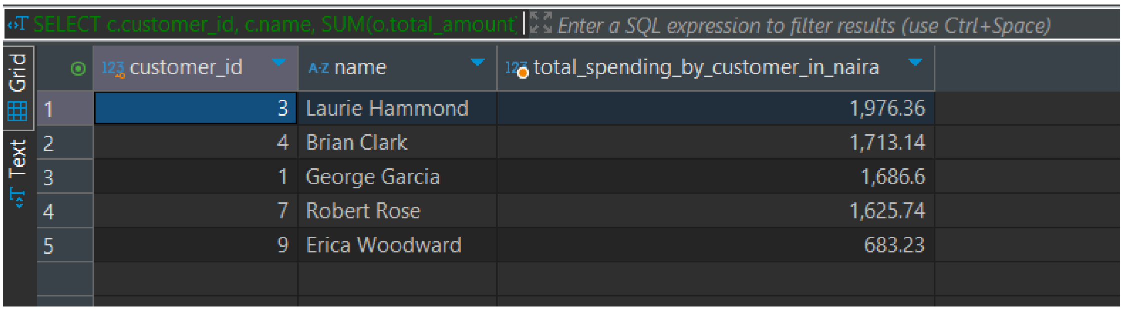
	product_id	product_name	total_revenue_by_product_in_naira
1	1	Wireless Earbuds	155.37
2	2	Office Chair	466.85
3	4	Men's Running Shoes	170.58
4	5	LED Desk Lamp	969.41
5	6	Acrylic Paint Set	163.26
6	7	Bluetooth Speaker	211.14
7	8	Laptop Backpack	772.2
8	9	Smartwatch	907.32
9	10	Yoga Mat	917.9
10	11	Electric Kettle	90.5
11	12	Gaming Keyboard	69.99
12	13	Wooden Coffee Table	241
13	14	Cotton Bedsheet Set	35.4
14	16	Women's Handbag	449.95
15	17	Mountain Bike	2,399.92
16	18	Portable Power Bank	359.91
17	19	Winter Jacket	250
18	20	Noise-Cancelling Headphones	1,349.91

A breakdown of revenue by product revealed how specific items contributed to overall earnings, with ORDER BY used to rank products.

2. Customer Insights:

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

```
SELECT
	c.customer_id,
	c.name,
	SUM(o.total_amount) AS total_spending_by_customer_in_naira
FROM
	orders o
JOIN customers c ON
	(o.customer_id = c.customer_id)
GROUP BY c.customer_id, c.name
ORDER BY total_spending_by_customer_in_naira DESC LIMIT 5;
```



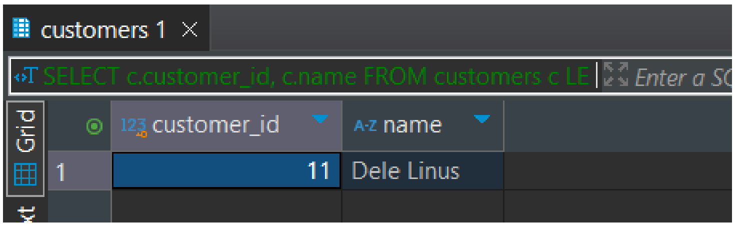
SELECT c.customer_id, c.name, SUM(o.total_amount | Enter a SQL expression to filter results (use Ctrl+Space)

	customer_id	name	total_spending_by_customer_in_naira
1	3	Laurie Hammond	1,976.36
2	4	Brian Clark	1,713.14
3	1	George Garcia	1,686.6
4	7	Robert Rose	1,625.74
5	9	Erica Woodward	683.23

The analusis identified the top 5 customers by their total spending, using SUM(total_amount) grouped by customer_id and ranked in descending order.

- **Identify customers who haven't made any purchases.**

```sql
SELECT
	c.customer_id,
	c.name
FROM
	customers c
LEFT JOIN orders o ON
	(o.customer_id = c.customer_id)
WHERE
	o.order_id IS NULL;
```



SELECT c.customer_id, c.name FROM customers c LE

	customer_id	name
1	11	Dele Linus

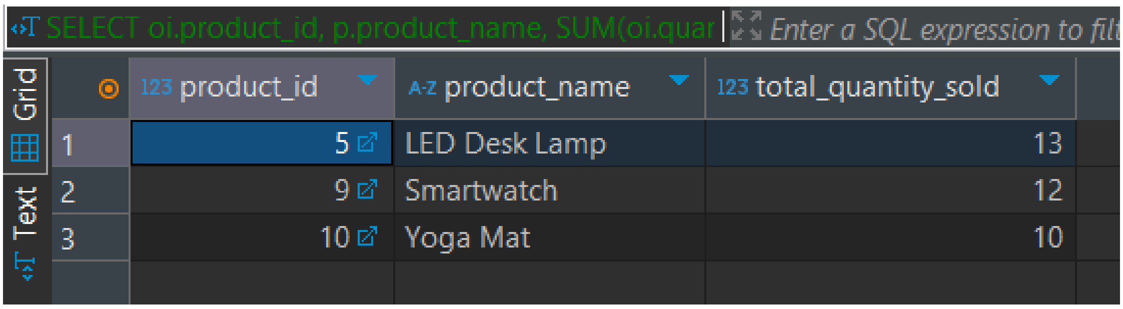
"Dele Linus" who haven't placed any orders was identified through a LEFT JOIN.

3. Product Trends:

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

The top 3 best-selling products were highlighted by summing quantities sold from the order_items table and ranking products by sales volume.

```sql
SELECT
	oi.product_id,
	p.product_name,
	SUM(oi.quantity) AS total_quantity_sold
FROM
	order_items oi
JOIN products p ON
	(oi.product_id = p.product_id)
GROUP BY
	oi.product_id,
	p.product_name
ORDER BY
	total_quantity_sold DESC
LIMIT 3;
```

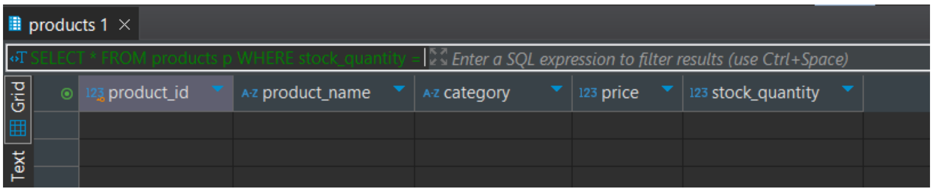


SELECT oi.product_id, p.product_name, SUM(oi.quar

	product_id	product_name	total_quantity_sold
1	5	LED Desk Lamp	13
2	9	Smartwatch	12
3	10	Yoga Mat	10

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

```
SELECT * FROM products p WHERE stock_quantity = 0;
```



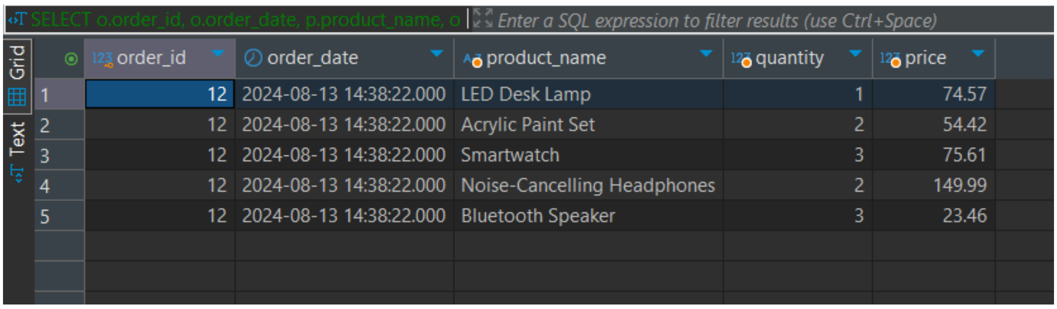
SELECT * FROM products p WHERE stock_quantity =

product_id	product_name	category	price	stock_quantity

4. Order Details:

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

```
SELECT
	o.order_id,
	o.order_date,
	p.product_name,
	oi.quantity,
	oi.price
FROM
	order_items oi
JOIN orders o ON
	(oi.order_id = o.order_id)
JOIN products p ON
	(oi.product_id = p.product_id)
WHERE
	o.order_id = 12;
```

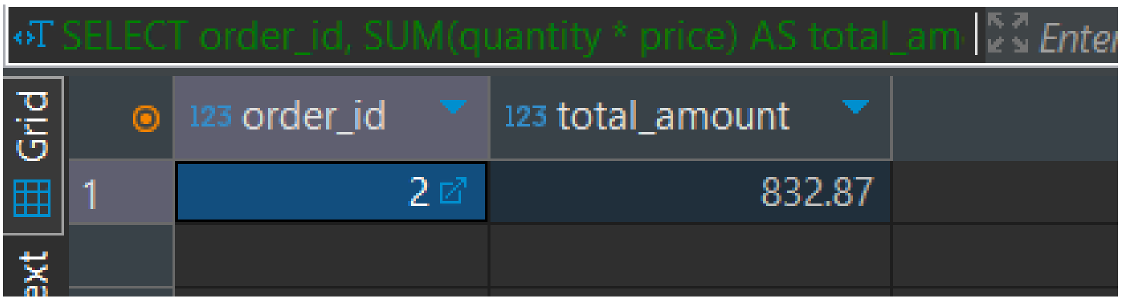


SELECT o.order_id, o.order_date, p.product_name, o

	order_id	order_date	product_name	quantity	price
1	12	2024-08-13 14:38:22.000	LED Desk Lamp	1	74.57
2	12	2024-08-13 14:38:22.000	Acrylic Paint Set	2	54.42
3	12	2024-08-13 14:38:22.000	Smartwatch	3	75.61
4	12	2024-08-13 14:38:22.000	Noise-Cancelling Headphones	2	149.99
5	12	2024-08-13 14:38:22.000	Bluetooth Speaker	3	23.46

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

```
SELECT
	order_id,
	SUM(quantity * price) AS total_amount
FROM
	order_items
GROUP BY
	order_id
HAVING
	order_id = 2;
```



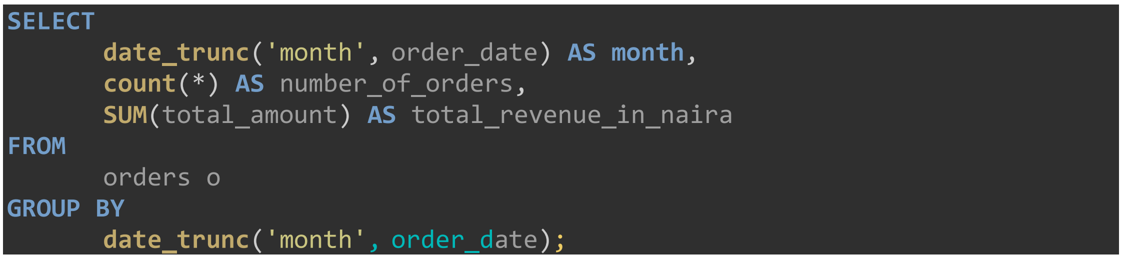
SELECT order_id, SUM(quantity * price) AS total_am

	order_id	total_amount
1	2	832.87

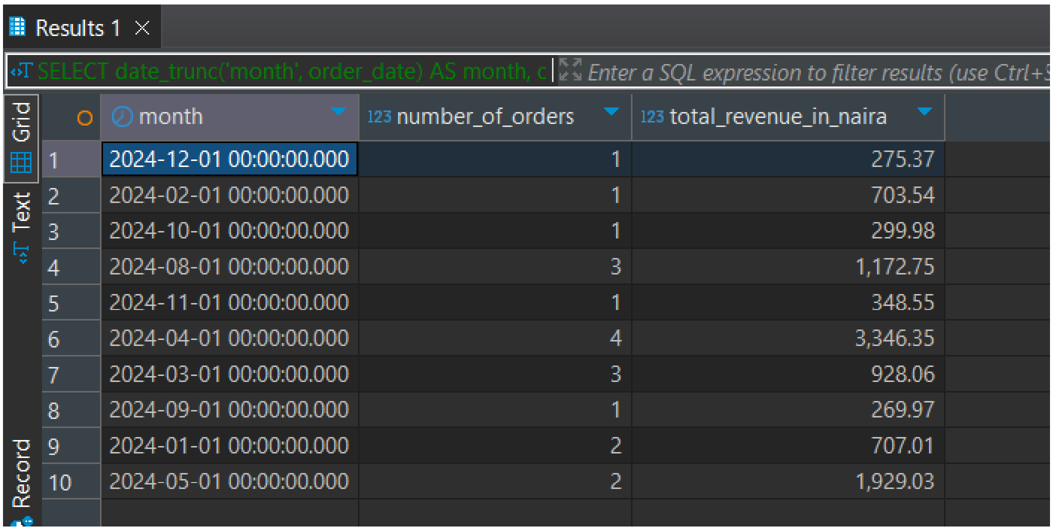
5. **Monthly Trends:**

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

This analyzed the orders and revenue trends monthly, using date_trunc(‘month’) to group by months, providing insights into seasonality and sales patterns.



```
SELECT
	date_trunc('month', order_date) AS month,
	count(*) AS number_of_orders,
	SUM(total_amount) AS total_revenue_in_naira
FROM
	orders o
GROUP BY
	date_trunc('month', order_date);
```



SELECT date_trunc('month', order_date) AS month, c

Enter a SQL expression to filter results (use Ctrl+S

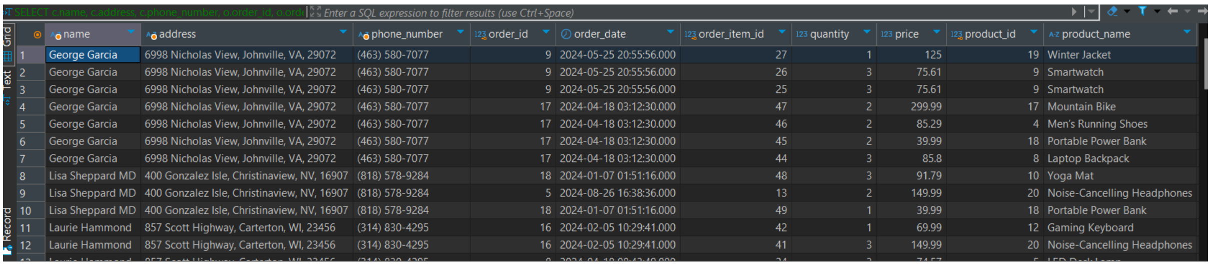
	month	number_of_orders	total_revenue_in_naira
1	2024-12-01 00:00:00.000	1	275.37
2	2024-02-01 00:00:00.000	1	703.54
3	2024-10-01 00:00:00.000	1	299.98
4	2024-08-01 00:00:00.000	3	1,172.75
5	2024-11-01 00:00:00.000	1	348.55
6	2024-04-01 00:00:00.000	4	3,346.35
7	2024-03-01 00:00:00.000	3	928.06
8	2024-09-01 00:00:00.000	1	269.97
9	2024-01-01 00:00:00.000	2	707.01
10	2024-05-01 00:00:00.000	2	1,929.03

Analytical Queries

1. **Joins: Write queries using INNER JOIN, LEFT JOIN, and FULL JOIN to retrieve data across multiple tables**

- **Using INNER JOIN to return only customers that ordered and details of their ordered items**

```sql
SELECT
        c.name, c.address, c.phone_number,
        o.order_id, o.order_date,
        oi.order_item_id, oi.quantity, oi.price,
        p.product_id, p.product_name
FROM
        customers c
JOIN orders o ON
        (C.customer_id = O.customer_id)
JOIN order_items oi ON
        (o.order_id = oi.order_id)
JOIN products p ON
        (oi.product_id = p.product_id)
ORDER BY
        c.customer_id;
```



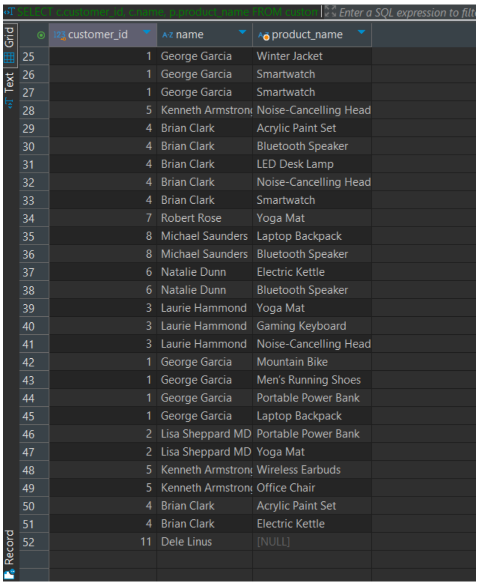
SELECT c.name, c.address, c.phone_number, o.order_id, o.ord

Enter a SQL expression to filter results (use Ctrl+Space)

	name	address	phone_number	order_id	order_date	order_item_id	quantity	price	product_id	product_name
1	George Garcia	6998 Nicholas View, Johnville, VA, 29072	(463) 580-7077	9	2024-05-25 20:55:56.000	27	1	125	19	Winter Jacket
2	George Garcia	6998 Nicholas View, Johnville, VA, 29072	(463) 580-7077	9	2024-05-25 20:55:56.000	26	3	75.61	9	Smartwatch
3	George Garcia	6998 Nicholas View, Johnville, VA, 29072	(463) 580-7077	9	2024-05-25 20:55:56.000	25	3	75.61	9	Smartwatch
4	George Garcia	6998 Nicholas View, Johnville, VA, 29072	(463) 580-7077	17	2024-04-18 03:12:30.000	47	2	299.99	17	Mountain Bike
5	George Garcia	6998 Nicholas View, Johnville, VA, 29072	(463) 580-7077	17	2024-04-18 03:12:30.000	46	2	85.29	4	Men's Running Shoes
6	George Garcia	6998 Nicholas View, Johnville, VA, 29072	(463) 580-7077	17	2024-04-18 03:12:30.000	45	2	39.99	18	Portable Power Bank
7	George Garcia	6998 Nicholas View, Johnville, VA, 29072	(463) 580-7077	17	2024-04-18 03:12:30.000	44	3	85.8	8	Laptop Backpack
8	Lisa Sheppard MD	400 Gonzalez Isle, Christinaview, NV, 16907	(818) 578-9284	18	2024-01-07 01:51:16.000	48	3	91.79	10	Yoga Mat
9	Lisa Sheppard MD	400 Gonzalez Isle, Christinaview, NV, 16907	(818) 578-9284	5	2024-08-26 16:38:36.000	13	2	149.99	20	Noise-Cancelling Headphones
10	Lisa Sheppard MD	400 Gonzalez Isle, Christinaview, NV, 16907	(818) 578-9284	18	2024-01-07 01:51:16.000	49	1	39.99	18	Portable Power Bank
11	Laurie Hammond	857 Scott Highway, Carterton, WI, 23456	(314) 830-4295	16	2024-02-05 10:29:41.000	42	1	69.99	12	Gaming Keyboard
12	Laurie Hammond	857 Scott Highway, Carterton, WI, 23456	(314) 830-4295	16	2024-02-05 10:29:41.000	41	3	149.99	20	Noise-Cancelling Headphones

- **Using LEFT JOIN to return customers and products they ordered whether or not they made an order**

```sql
SELECT
        c.customer_id, c.name, p.product_name
FROM
        customers c
LEFT JOIN orders o ON
        (C.customer_id = O.customer_id)
LEFT JOIN order_items oi ON
        (o.order_id = oi.order_id)
LEFT JOIN products p ON
        (oi.product_id = p.product_id);
```

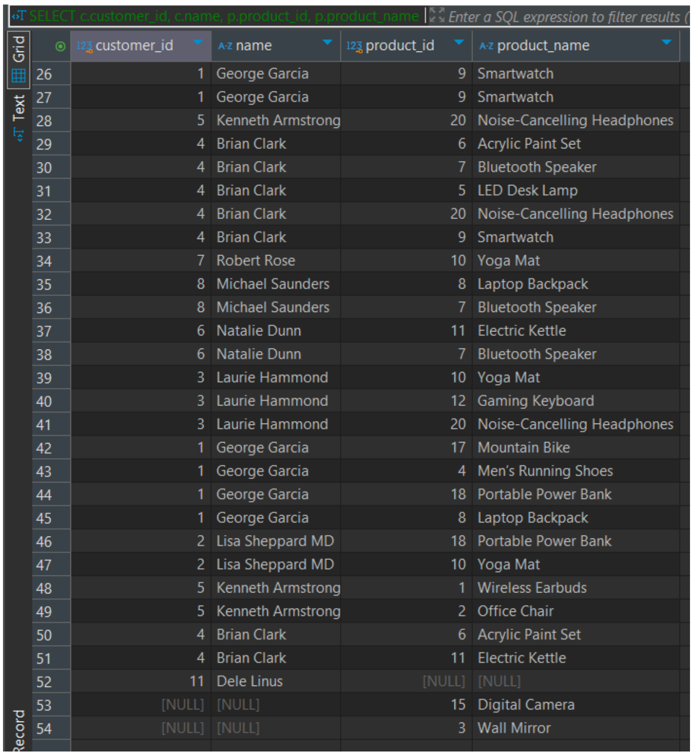


SELECT c.customer_id, c.name, p.product_name FROM custom

	customer_id	name	product_name
25	1	George Garcia	Winter Jacket
26	1	George Garcia	Smartwatch
27	1	George Garcia	Smartwatch
28	5	Kenneth Armstronç	Noise-Cancelling Head
29	4	Brian Clark	Acrylic Paint Set
30	4	Brian Clark	Bluetooth Speaker
31	4	Brian Clark	LED Desk Lamp
32	4	Brian Clark	Noise-Cancelling Head
33	4	Brian Clark	Smartwatch
34	7	Robert Rose	Yoga Mat
35	8	Michael Saunders	Laptop Backpack
36	8	Michael Saunders	Bluetooth Speaker
37	6	Natalie Dunn	Electric Kettle
38	6	Natalie Dunn	Bluetooth Speaker
39	3	Laurie Hammond	Yoga Mat
40	3	Laurie Hammond	Gaming Keyboard
41	3	Laurie Hammond	Noise-Cancelling Head
42	1	George Garcia	Mountain Bike
43	1	George Garcia	Men's Running Shoes
44	1	George Garcia	Portable Power Bank
45	1	George Garcia	Laptop Backpack
46	2	Lisa Sheppard MD	Portable Power Bank
47	2	Lisa Sheppard MD	Yoga Mat
48	5	Kenneth Armstronç	Wireless Earbuds
49	5	Kenneth Armstronç	Office Chair
50	4	Brian Clark	Acrylic Paint Set
51	4	Brian Clark	Electric Kettle
52	11	Dele Linus	[NULL]

- **Using FULL JOIN to return customers and products whether or not they are matched in orders.**

```
SELECT
      c.customer_id, c.name, p.product_id, p.product_name
FROM
      customers c
FULL JOIN orders o ON (C.customer_id = O.customer_id)
FULL JOIN order_items oi ON (o.order_id = oi.order_id)
FULL JOIN products p ON (oi.product_id = p.product_id);
```



SELECT c.customer_id, c.name, p.product_id, p.product_name

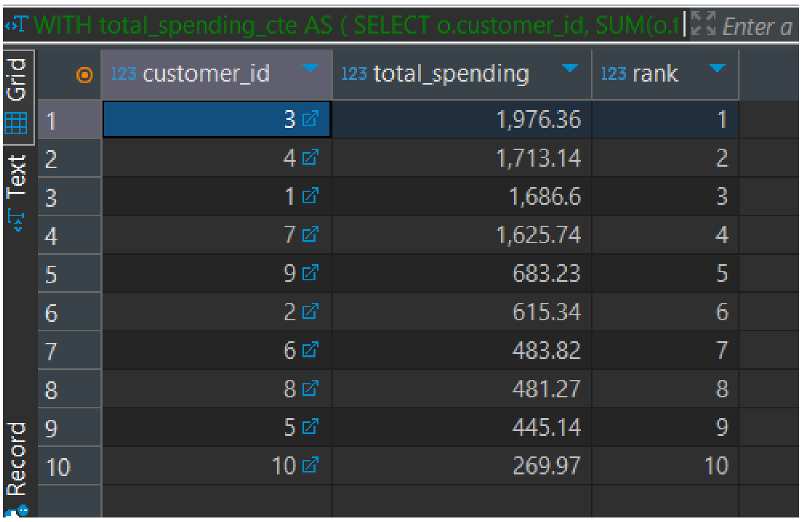
	customer_id	name	product_id	product_name
26	1	George Garcia	9	Smartwatch
27	1	George Garcia	9	Smartwatch
28	5	Kenneth Armstrong	20	Noise-Cancelling Headphones
29	4	Brian Clark	6	Acrylic Paint Set
30	4	Brian Clark	7	Bluetooth Speaker
31	4	Brian Clark	5	LED Desk Lamp
32	4	Brian Clark	20	Noise-Cancelling Headphones
33	4	Brian Clark	9	Smartwatch
34	7	Robert Rose	10	Yoga Mat
35	8	Michael Saunders	8	Laptop Backpack
36	8	Michael Saunders	7	Bluetooth Speaker
37	6	Natalie Dunn	11	Electric Kettle
38	6	Natalie Dunn	7	Bluetooth Speaker
39	3	Laurie Hammond	10	Yoga Mat
40	3	Laurie Hammond	12	Gaming Keyboard
41	3	Laurie Hammond	20	Noise-Cancelling Headphones
42	1	George Garcia	17	Mountain Bike
43	1	George Garcia	4	Men's Running Shoes
44	1	George Garcia	18	Portable Power Bank
45	1	George Garcia	8	Laptop Backpack
46	2	Lisa Sheppard MD	18	Portable Power Bank
47	2	Lisa Sheppard MD	10	Yoga Mat
48	5	Kenneth Armstrong	1	Wireless Earbuds
49	5	Kenneth Armstrong	2	Office Chair
50	4	Brian Clark	6	Acrylic Paint Set
51	4	Brian Clark	11	Electric Kettle
52	11	Dele Linus	[NULL]	[NULL]
53	[NULL]	[NULL]	15	Digital Camera
54	[NULL]	[NULL]	3	Wall Mirror

2. Window Functions:

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

```
WITH total_spending_cte AS (
SELECT
	o.customer_id,
	SUM(o.total_amount) AS total_spending
FROM
	orders o GROUP BY o.customer_id ORDER BY o.customer_id)

SELECT *, RANK() OVER (ORDER BY total_spending DESC)
FROM total_spending_cte;
```

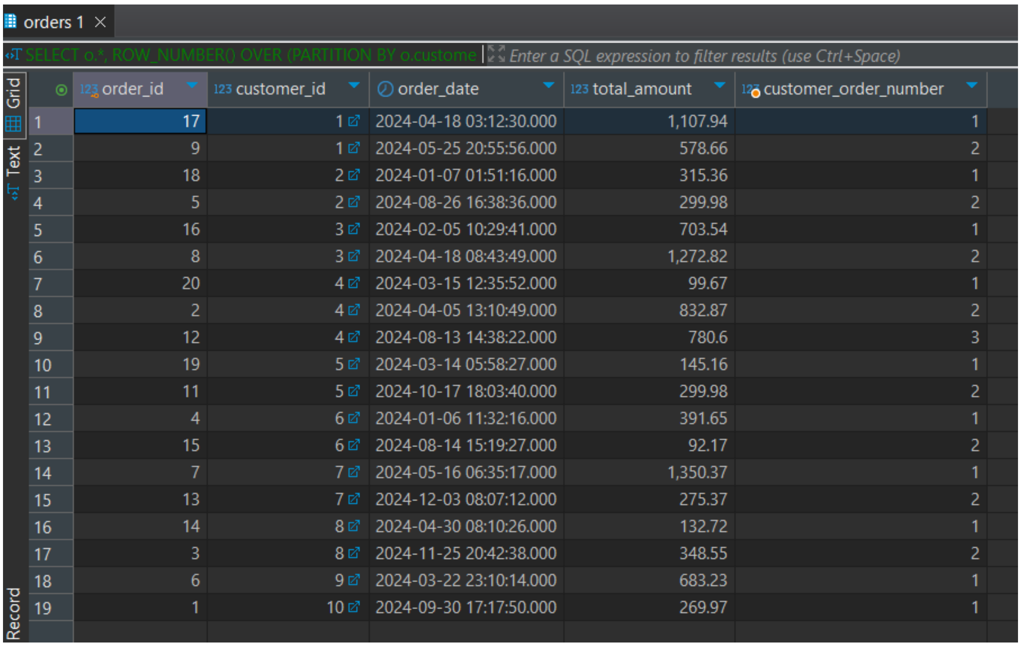


WITH total_spending_cte AS (SELECT o.customer_id, SUM(o.t

	customer_id	total_spending	rank
1	3	1,976.36	1
2	4	1,713.14	2
3	1	1,686.6	3
4	7	1,625.74	4
5	9	683.23	5
6	2	615.34	6
7	6	483.82	7
8	8	481.27	8
9	5	445.14	9
10	10	269.97	10

- **Use ROW_NUMBER() to assign a unique number to each order for a customer**

```
SELECT
	o.*, ROW_NUMBER() OVER (PARTITION BY o.customer_id
ORDER BY o.order_date) AS customer_order_number
FROM orders o;
```



SELECT o.*, ROW_NUMBER() OVER (PARTITION BY o.custome

Enter a SQL expression to filter results (use Ctrl+Space)

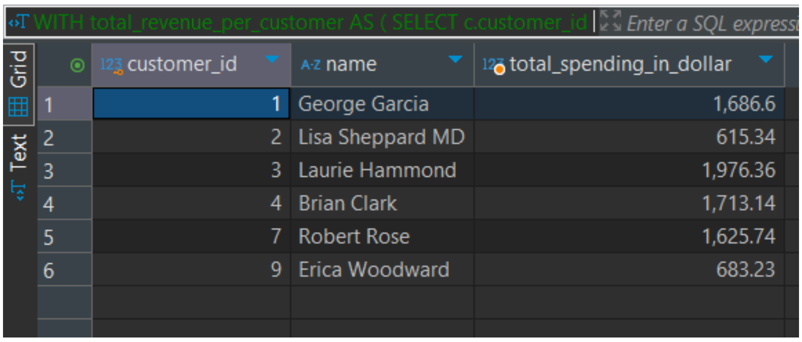
	order_id	customer_id	order_date	total_amount	customer_order_number
1	17	1	2024-04-18 03:12:30.000	1,107.94	1
2	9	1	2024-05-25 20:55:56.000	578.66	2
3	18	2	2024-01-07 01:51:16.000	315.36	1
4	5	2	2024-08-26 16:38:36.000	299.98	2
5	16	3	2024-02-05 10:29:41.000	703.54	1
6	8	3	2024-04-18 08:43:49.000	1,272.82	2
7	20	4	2024-03-15 12:35:52.000	99.67	1
8	2	4	2024-04-05 13:10:49.000	832.87	2
9	12	4	2024-08-13 14:38:22.000	780.6	3
10	19	5	2024-03-14 05:58:27.000	145.16	1
11	11	5	2024-10-17 18:03:40.000	299.98	2
12	4	6	2024-01-06 11:32:16.000	391.65	1
13	15	6	2024-08-14 15:19:27.000	92.17	2
14	7	7	2024-05-16 06:35:17.000	1,350.37	1
15	13	7	2024-12-03 08:07:12.000	275.37	2
16	14	8	2024-04-30 08:10:26.000	132.72	1
17	3	8	2024-11-25 20:42:38.000	348.55	2
18	6	9	2024-03-22 23:10:14.000	683.23	1
19	1	10	2024-09-30 17:17:50.000	269.97	1

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 total_revenue_per_customer AS (
      SELECT c.customer_id, c.name, SUM(o.total_amount) AS
total_spending_in_dollar
      FROM orders o
      JOIN customers c ON (o.customer_id = c.customer_id)
      GROUP BY c.customer_id, c.name)

            SELECT * FROM total_revenue_per_customer WHERE
            total_spending_in_dollar > 500 ORDER BY customer_id;
```



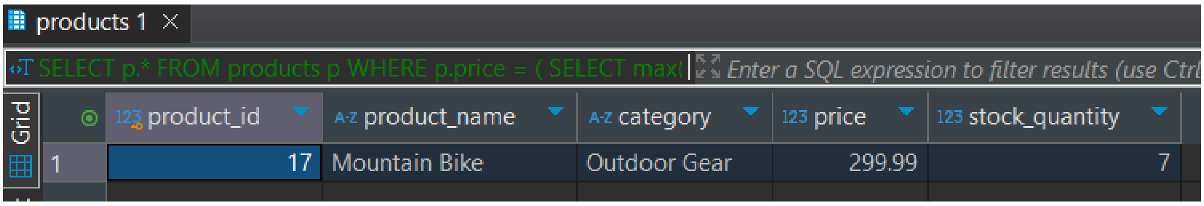
WITH total_revenue_per_customer AS (SELECT c.customer_id

Enter a SQL express

	customer_id	name	total_spending_in_dollar
1	1	George Garcia	1,686.6
2	2	Lisa Sheppard MD	615.34
3	3	Laurie Hammond	1,976.36
4	4	Brian Clark	1,713.14
5	7	Robert Rose	1,625.74
6	9	Erica Woodward	683.23

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

```
SELECT p.* FROM products p WHERE p.price = (
       SELECT max(p2.price) FROM products p2);
```


products 1
SELECT p.* FROM products p WHERE p.price = (SELECT max(
Enter a SQL expression to filter results (use Ctrl
Grid

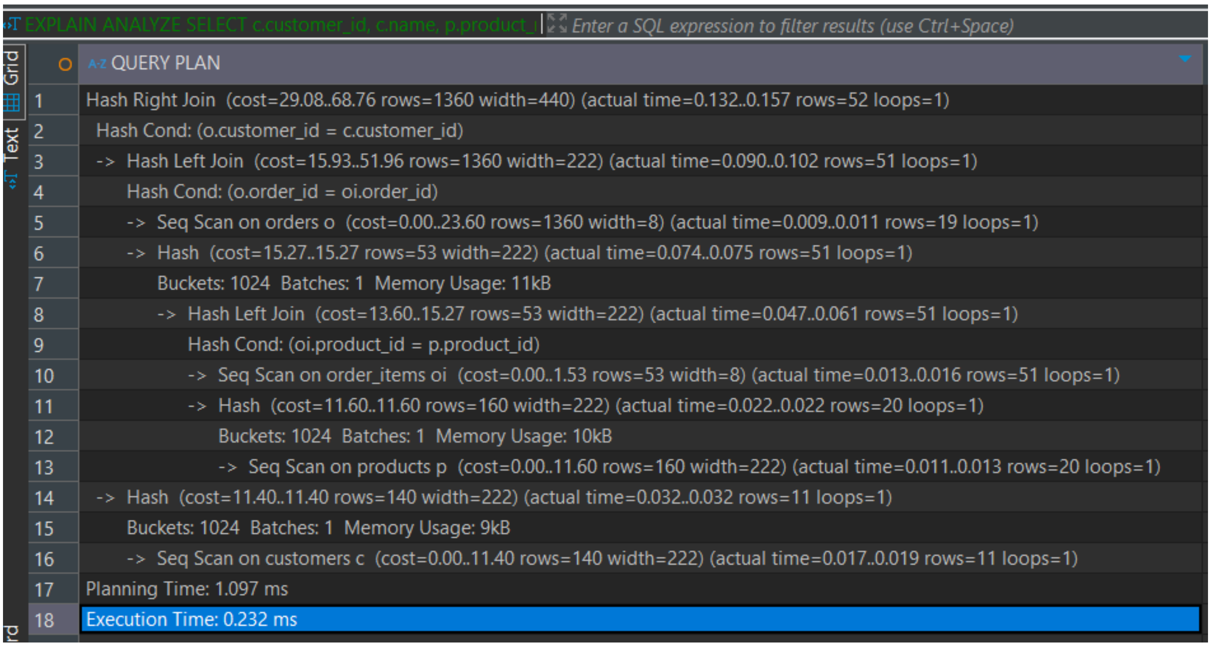
	product_id	product_name	category	price	stock_quantity
1	17	Mountain Bike	Outdoor Gear	299.99	7

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.

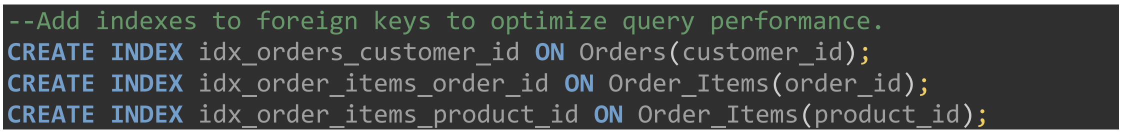
- **Performance before Indexing**

```
-- Before
EXPLAIN ANALYZE SELECT
      c.customer_id, c.name, p.product_name
FROM customers c
LEFT JOIN orders o ON (C.customer_id = O.customer_id)
LEFT JOIN order_items oi ON (o.order_id = oi.order_id)
LEFT JOIN products p ON (oi.product_id = p.product_id);
```

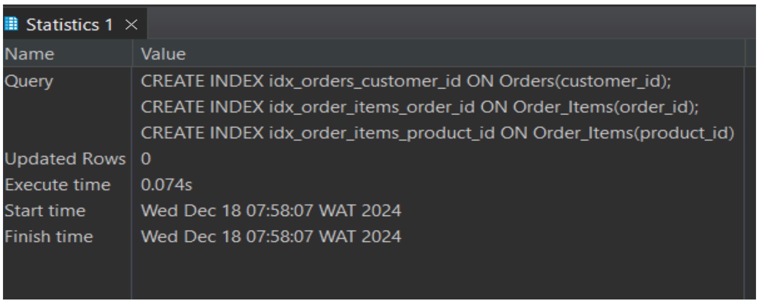

EXPLAIN ANALYZE SELECT c.customer_id, c.name, p.product_i
Enter a SQL expression to filter results (use Ctrl+Space)
Grid
Text
QUERY PLAN

```
Hash Right Join  (cost=29.08..68.76 rows=1360 width=440) (actual time=0.132..0.157 rows=52 loops=1)
  Hash Cond: (o.customer_id = c.customer_id)
  ->  Hash Left Join  (cost=15.93..51.96 rows=1360 width=222) (actual time=0.090..0.102 rows=51 loops=1)
        Hash Cond: (o.order_id = oi.order_id)
        ->  Seq Scan on orders o  (cost=0.00..23.60 rows=1360 width=8) (actual time=0.009..0.011 rows=19 loops=1)
        ->  Hash  (cost=15.27..15.27 rows=53 width=222) (actual time=0.074..0.075 rows=51 loops=1)
              Buckets: 1024  Batches: 1  Memory Usage: 11kB
              ->  Hash Left Join  (cost=13.60..15.27 rows=53 width=222) (actual time=0.047..0.061 rows=51 loops=1)
                    Hash Cond: (oi.product_id = p.product_id)
                    ->  Seq Scan on order_items oi  (cost=0.00..1.53 rows=53 width=8) (actual time=0.013..0.016 rows=51 loops=1)
                    ->  Hash  (cost=11.60..11.60 rows=160 width=222) (actual time=0.022..0.022 rows=20 loops=1)
                          Buckets: 1024  Batches: 1  Memory Usage: 10kB
                          ->  Seq Scan on products p  (cost=0.00..11.60 rows=160 width=222) (actual time=0.011..0.013 rows=20 loops=1)
  ->  Hash  (cost=11.40..11.40 rows=140 width=222) (actual time=0.032..0.032 rows=11 loops=1)
        Buckets: 1024  Batches: 1  Memory Usage: 9kB
        ->  Seq Scan on customers c  (cost=0.00..11.40 rows=140 width=222) (actual time=0.017..0.019 rows=11 loops=1)
Planning Time: 1.097 ms
Execution Time: 0.232 ms
```

- **Indexing the foreign keys.**



```
--Add indexes to foreign keys to optimize query performance.
CREATE INDEX idx_orders_customer_id ON Orders(customer_id);
CREATE INDEX idx_order_items_order_id ON Order_Items(order_id);
CREATE INDEX idx_order_items_product_id ON Order_Items(product_id);
```

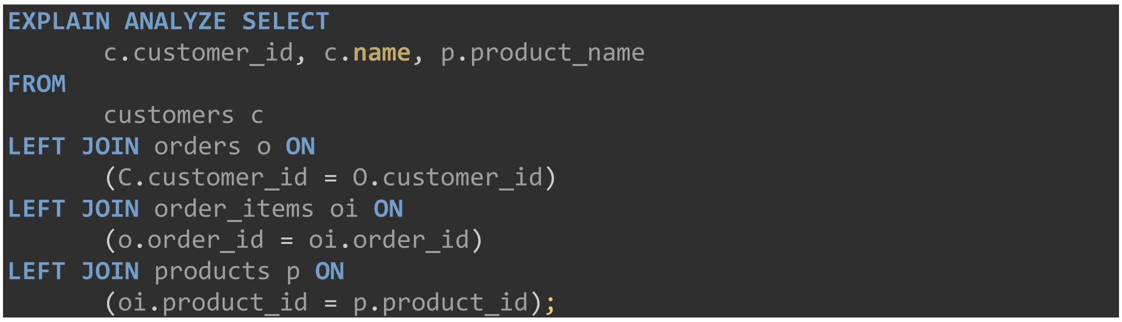


Name	Value

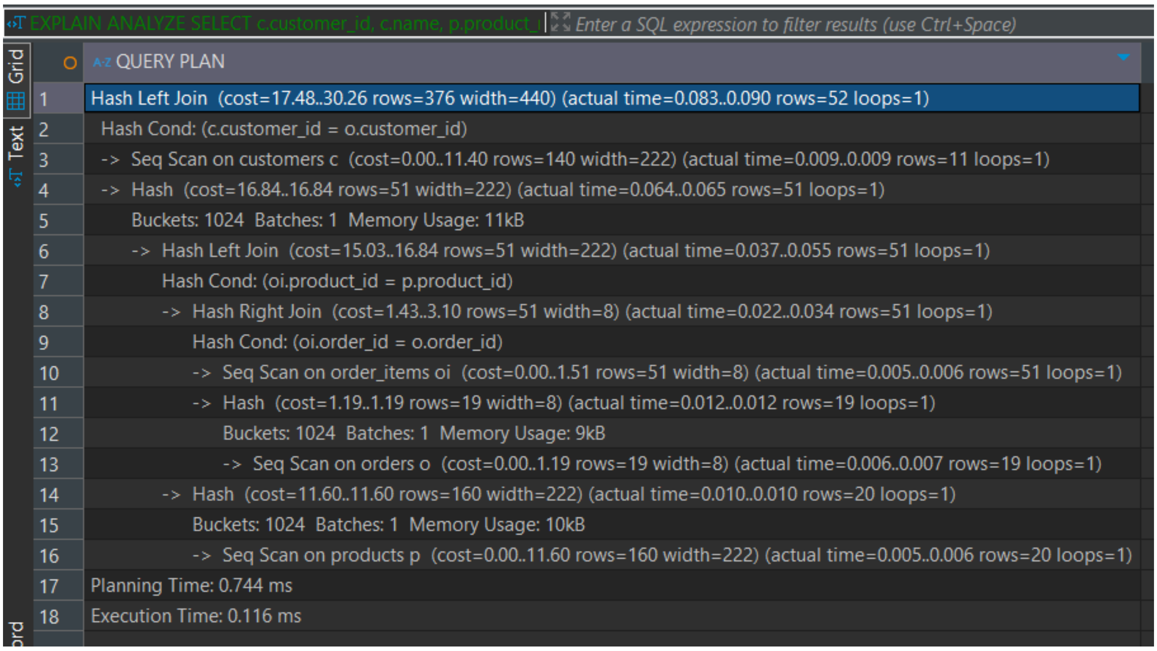
Query	CREATE INDEX idx_orders_customer_id ON Orders(customer_id);
	CREATE INDEX idx_order_items_order_id ON Order_Items(order_id);
	CREATE INDEX idx_order_items_product_id ON Order_Items(product_id)

Updated Rows	0
Execute time	0.074s
Start time	Wed Dec 18 07:58:07 WAT 2024
Finish time	Wed Dec 18 07:58:07 WAT 2024

- **Performance after indexing.**



```
EXPLAIN ANALYZE SELECT
       c.customer_id, c.name, p.product_name
FROM
       customers c
LEFT JOIN orders o ON
       (C.customer_id = O.customer_id)
LEFT JOIN order_items oi ON
       (o.order_id = oi.order_id)
LEFT JOIN products p ON
       (oi.product_id = p.product_id);
```



EXPLAIN ANALYZE SELECT c.customer_id, c.name, p.product_i
Enter a SQL expression to filter results (use Ctrl+Space)
Grid
Text

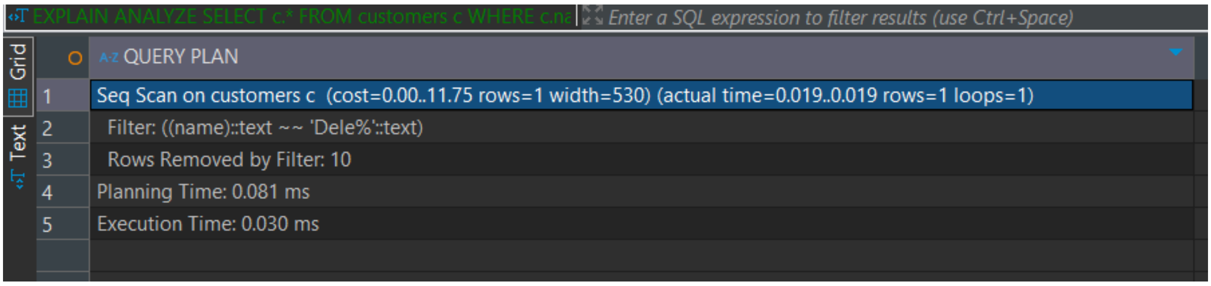
```
QUERY PLAN
Hash Left Join  (cost=17.48..30.26 rows=376 width=440) (actual time=0.083..0.090 rows=52 loops=1)
  Hash Cond: (c.customer_id = o.customer_id)
  ->  Seq Scan on customers c  (cost=0.00..11.40 rows=140 width=222) (actual time=0.009..0.009 rows=11 loops=1)
  ->  Hash  (cost=16.84..16.84 rows=51 width=222) (actual time=0.064..0.065 rows=51 loops=1)
        Buckets: 1024  Batches: 1  Memory Usage: 11kB
        ->  Hash Left Join  (cost=15.03..16.84 rows=51 width=222) (actual time=0.037..0.055 rows=51 loops=1)
              Hash Cond: (oi.product_id = p.product_id)
              ->  Hash Right Join  (cost=1.43..3.10 rows=51 width=8) (actual time=0.022..0.034 rows=51 loops=1)
                    Hash Cond: (oi.order_id = o.order_id)
                    ->  Seq Scan on order_items oi  (cost=0.00..1.51 rows=51 width=8) (actual time=0.005..0.006 rows=51 loops=1)
                    ->  Hash  (cost=1.19..1.19 rows=19 width=8) (actual time=0.012..0.012 rows=19 loops=1)
                          Buckets: 1024  Batches: 1  Memory Usage: 9kB
                          ->  Seq Scan on orders o  (cost=0.00..1.19 rows=19 width=8) (actual time=0.006..0.007 rows=19 loops=1)
              ->  Hash  (cost=11.60..11.60 rows=160 width=222) (actual time=0.010..0.010 rows=20 loops=1)
                    Buckets: 1024  Batches: 1  Memory Usage: 10kB
                    ->  Seq Scan on products p  (cost=0.00..11.60 rows=160 width=222) (actual time=0.005..0.006 rows=20 loops=1)
Planning Time: 0.744 ms
Execution Time: 0.116 ms
```

It could be seen that the **execution time** and **cost** all got lower after indexing

5. Optimization:

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

```sql
EXPLAIN ANALYZE SELECT c.* FROM customers c WHERE c.name LIKE 'Dele%';
```


EXPLAIN ANALYZE SELECT c.* FROM customers c WHERE c.na
Enter a SQL expression to filter results (use Ctrl+Space)
Grid
Text

	QUERY PLAN
1	Seq Scan on customers c (cost=0.00..11.75 rows=1 width=530) (actual time=0.019..0.019 rows=1 loops=1)
2	Filter: ((name)::text ~~ 'Dele%'::text)
3	Rows Removed by Filter: 10
4	Planning Time: 0.081 ms
5	Execution Time: 0.030 ms

The node type being **sequential scan** means the engine scanned through all rows to find the one that fulfilled the filter condition.

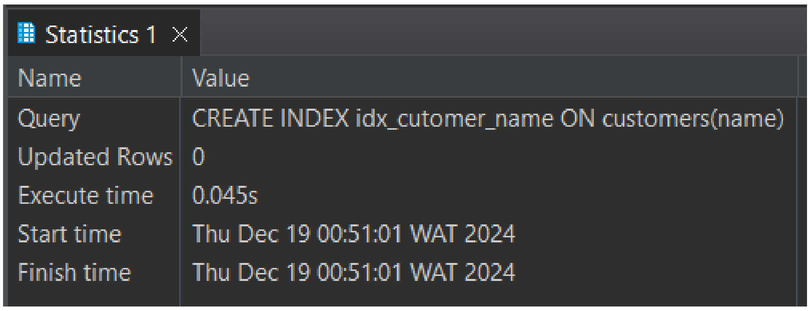
The **estimated costs** and **actual costs** seems to no align, as the actual costs appeared very far from the estimated costs.

There's only one **row** that fulfilled the query condition, while 10 rows were filtered out by the condition.

The time it took the query to run (**execution time**) was 0.030 milliseconds.

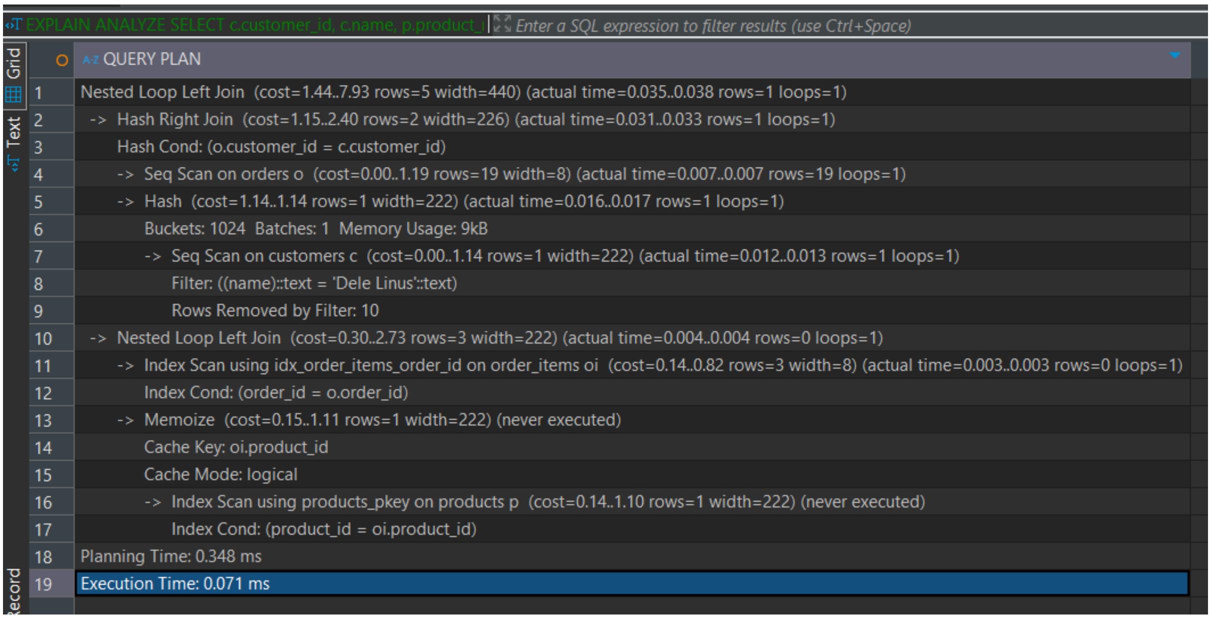
- **Optimize slow queries by adjusting indexes, reordering joins, or rewriting the query.**
  Index would be made on the name column where filter will be made.

```sql
CREATE INDEX idx_cutomer_name ON customers(name);
```



Name	Value
Query	CREATE INDEX idx_cutomer_name ON customers(name)
Updated Rows	0
Execute time	0.045s
Start time	Thu Dec 19 00:51:01 WAT 2024
Finish time	Thu Dec 19 00:51:01 WAT 2024

```sql
EXPLAIN ANALYZE SELECT
        c.customer_id, c.name, p.product_name
FROM customers c
LEFT JOIN orders o ON (C.customer_id = O.customer_id)
LEFT JOIN order_items oi ON (o.order_id = oi.order_id)
LEFT JOIN products p ON (oi.product_id = p.product_id)
WHERE c.name = 'Dele Linus';
```


EXPLAIN ANALYZE SELECT c.customer_id, c.name, p.product_i
Enter a SQL expression to filter results (use Ctrl+Space)
Grid
Text
Record
QUERY PLAN

```
Nested Loop Left Join  (cost=1.44..7.93 rows=5 width=440) (actual time=0.035..0.038 rows=1 loops=1)
  ->  Hash Right Join  (cost=1.15..2.40 rows=2 width=226) (actual time=0.031..0.033 rows=1 loops=1)
        Hash Cond: (o.customer_id = c.customer_id)
        ->  Seq Scan on orders o  (cost=0.00..1.19 rows=19 width=8) (actual time=0.007..0.007 rows=19 loops=1)
        ->  Hash  (cost=1.14..1.14 rows=1 width=222) (actual time=0.016..0.017 rows=1 loops=1)
              Buckets: 1024  Batches: 1  Memory Usage: 9kB
              ->  Seq Scan on customers c  (cost=0.00..1.14 rows=1 width=222) (actual time=0.012..0.013 rows=1 loops=1)
                    Filter: ((name)::text = 'Dele Linus'::text)
                    Rows Removed by Filter: 10
  ->  Nested Loop Left Join  (cost=0.30..2.73 rows=3 width=222) (actual time=0.004..0.004 rows=0 loops=1)
        ->  Index Scan using idx_order_items_order_id on order_items oi  (cost=0.14..0.82 rows=3 width=8) (actual time=0.003..0.003 rows=0 loops=1)
              Index Cond: (order_id = o.order_id)
        ->  Memoize  (cost=0.15..1.11 rows=1 width=222) (never executed)
              Cache Key: oi.product_id
              Cache Mode: logical
              ->  Index Scan using products_pkey on products p  (cost=0.14..1.10 rows=1 width=222) (never executed)
                    Index Cond: (product_id = oi.product_id)
Planning Time: 0.348 ms
Execution Time: 0.071 ms
```

It could be seen from the EXPLAIN ANALYZE result screenshot above that, filters and joins were done using index scan, which shows an optimization in the query performance.