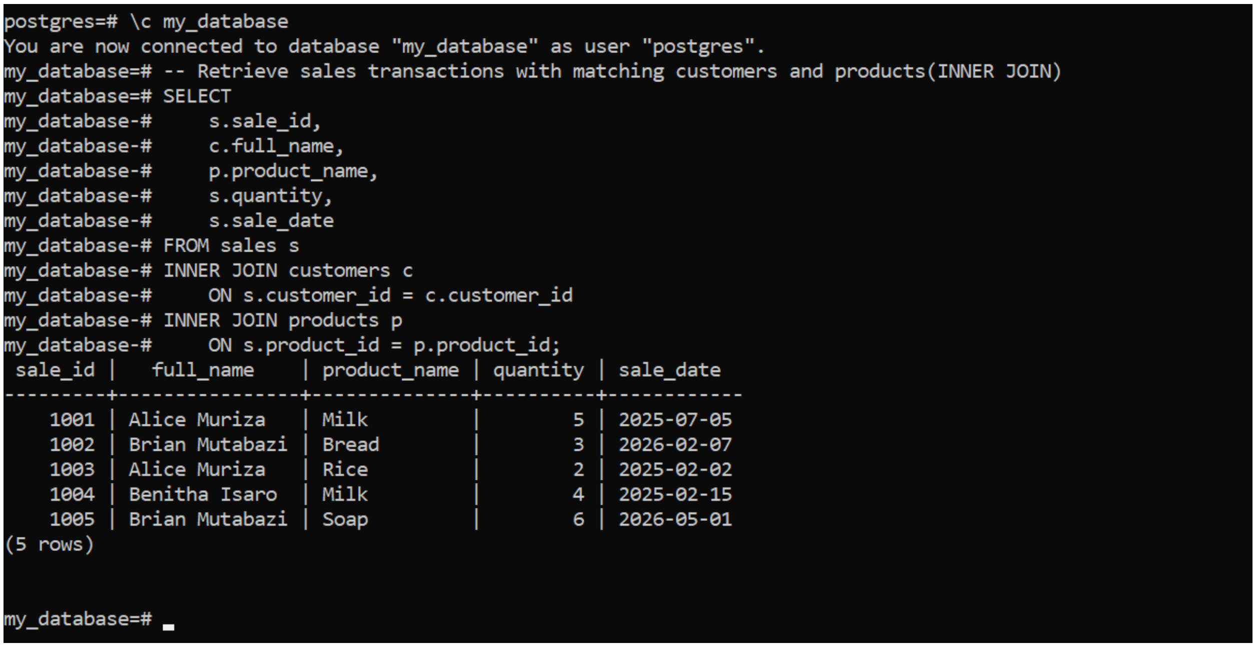
NAME: KARIZA PATIENCE
ID: 28795

JOIN SCREENSHOT

1. INNER JOIN

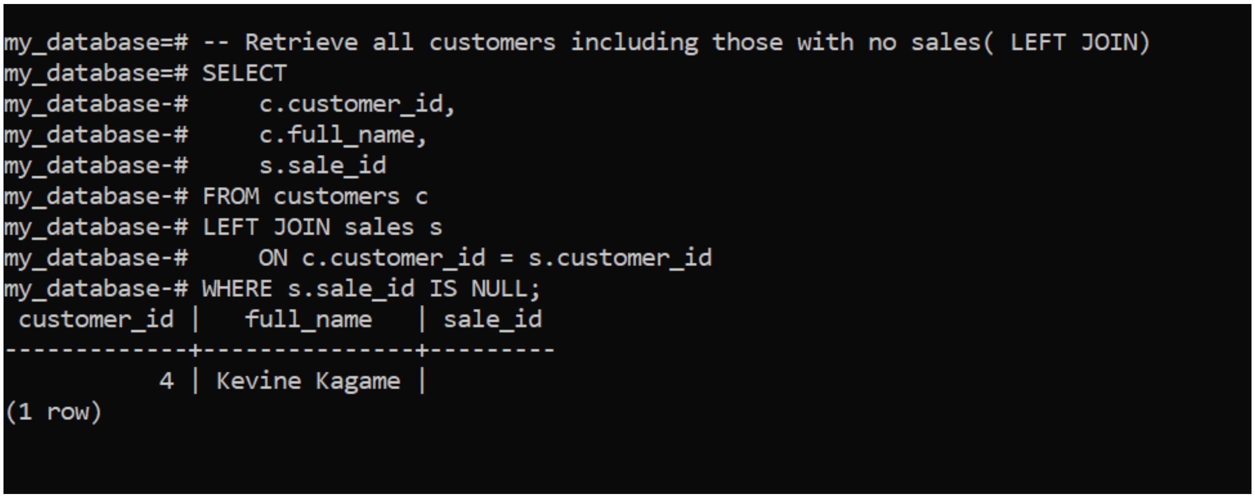


```
postgres=# \c my_database
You are now connected to database "my_database" as user "postgres".
my_database=# -- Retrieve sales transactions with matching customers and products(INNER JOIN)
my_database=# SELECT
my_database-#     s.sale_id,
my_database-#     c.full_name,
my_database-#     p.product_name,
my_database-#     s.quantity,
my_database-#     s.sale_date
my_database-# FROM sales s
my_database-# INNER JOIN customers c
my_database-#     ON s.customer_id = c.customer_id
my_database-# INNER JOIN products p
my_database-#     ON s.product_id = p.product_id;
 sale_id |   full_name    | product_name | quantity | sale_date
---------+----------------+--------------+----------+------------
    1001 | Alice Muriza   | Milk         |        5 | 2025-07-05
    1002 | Brian Mutabazi | Bread        |        3 | 2026-02-07
    1003 | Alice Muriza   | Rice         |        2 | 2025-02-02
    1004 | Benitha Isaro  | Milk         |        4 | 2025-02-15
    1005 | Brian Mutabazi | Soap         |        6 | 2026-05-01
(5 rows)


my_database=# _
```

Interpretation:
This query returns only sales records that have valid customer and product information. It helps the business analyze confirmed transactions and ensures data accuracy for revenue reporting.

2. LEFT JOIN

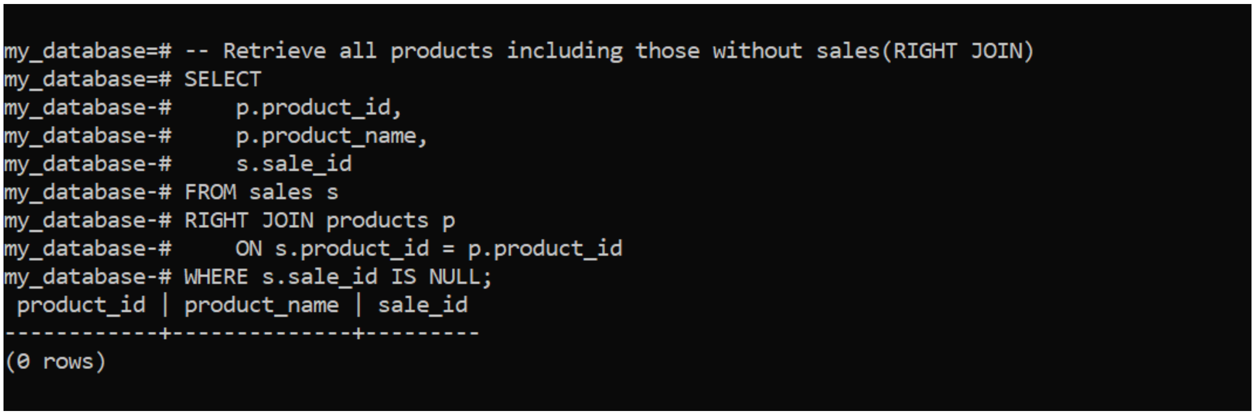


```
my_database=# -- Retrieve all customers including those with no sales( LEFT JOIN)
my_database=# SELECT
my_database-#     c.customer_id,
my_database-#     c.full_name,
my_database-#     s.sale_id
my_database-# FROM customers c
my_database-# LEFT JOIN sales s
my_database-#     ON c.customer_id = s.customer_id
my_database-# WHERE s.sale_id IS NULL;
 customer_id |   full_name   | sale_id
-------------+---------------+---------
           4 | Kevine Kagame |
(1 row)
```

Interpretation:

This query identifies customers who are registered but have never made a purchase. The business can target these customers with promotions or engagement campaigns to increase sales.

3. RIGHT JOIN

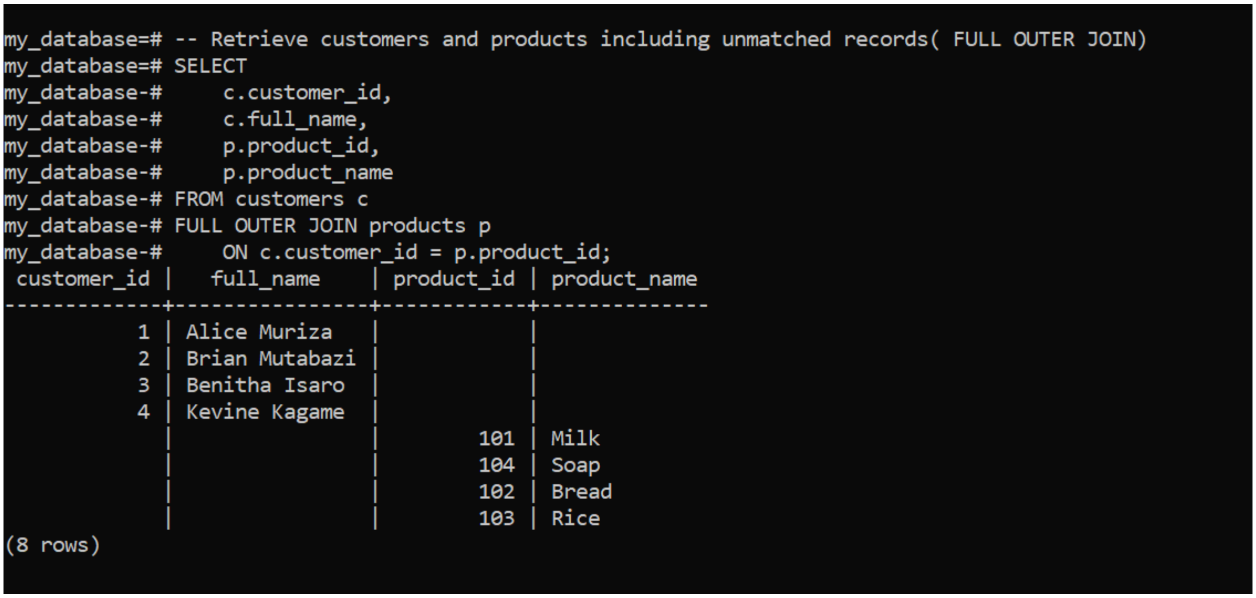


```
my_database=# -- Retrieve all products including those without sales(RIGHT JOIN)
my_database=# SELECT
my_database-#     p.product_id,
my_database-#     p.product_name,
my_database-#     s.sale_id
my_database-# FROM sales s
my_database-# RIGHT JOIN products p
my_database-#     ON s.product_id = p.product_id
my_database-# WHERE s.sale_id IS NULL;
 product_id | product_name | sale_id
------------+--------------+---------
(0 rows)
```

Interpretation:

This query highlights products that have never been sold. Management can decide whether to discontinue these products or introduce promotions to improve their performance.

4. FULL OUTER JOIN

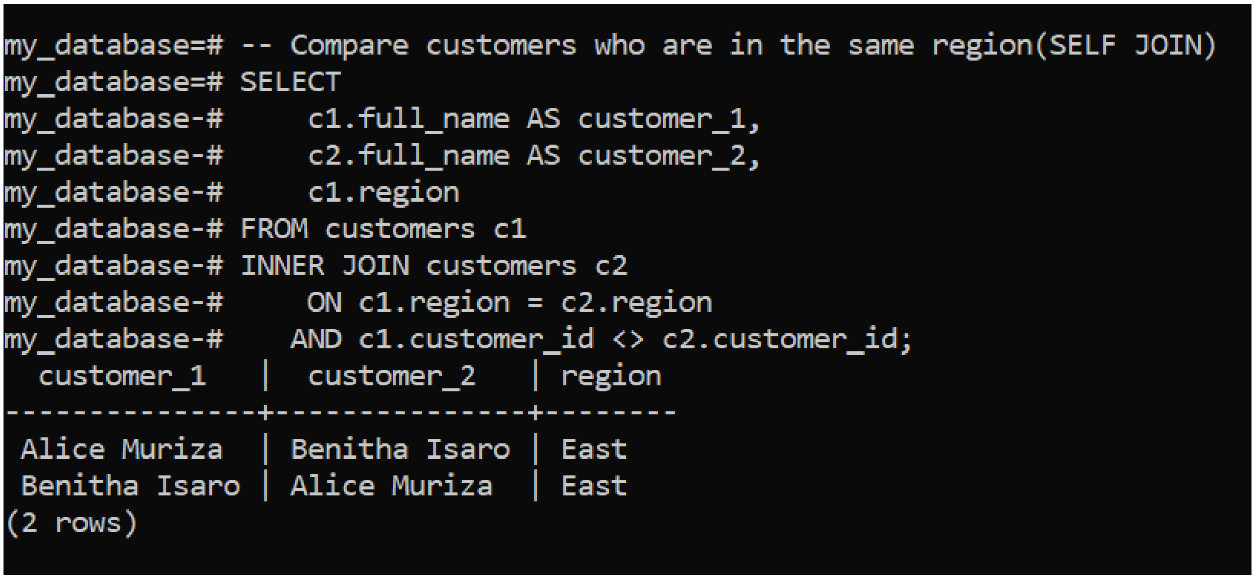


```
my_database=# -- Retrieve customers and products including unmatched records( FULL OUTER JOIN)
my_database=# SELECT
my_database-#     c.customer_id,
my_database-#     c.full_name,
my_database-#     p.product_id,
my_database-#     p.product_name
my_database-# FROM customers c
my_database-# FULL OUTER JOIN products p
my_database-#     ON c.customer_id = p.product_id;
 customer_id |   full_name    | product_id | product_name
-------------+----------------+------------+--------------
           1 | Alice Muriza   |            |
           2 | Brian Mutabazi |            |
           3 | Benitha Isaro  |            |
           4 | Kevine Kagame  |            |
             |                |        101 | Milk
             |                |        104 | Soap
             |                |        102 | Bread
             |                |        103 | Rice
(8 rows)
```

Interpretation:

This query includes all customers and all products, even when no direct relationship exists. It helps identify gaps in sales coverage and ensures a complete overview of both entities.

5. SELF JOIN



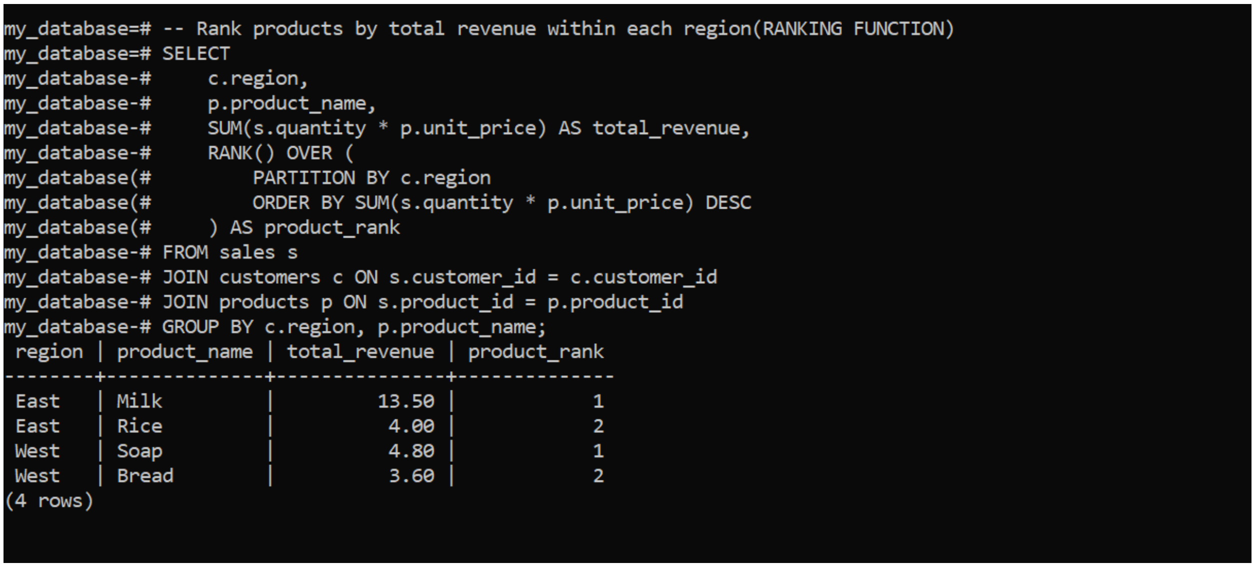
```
my_database=# -- Compare customers who are in the same region(SELF JOIN)
my_database=# SELECT
my_database-#     c1.full_name AS customer_1,
my_database-#     c2.full_name AS customer_2,
my_database-#     c1.region
my_database-# FROM customers c1
my_database-# INNER JOIN customers c2
my_database-#     ON c1.region = c2.region
my_database-#    AND c1.customer_id <> c2.customer_id;
  customer_1   |  customer_2   | region
---------------+---------------+--------
 Alice Muriza  | Benitha Isaro | East
 Benitha Isaro | Alice Muriza  | East
(2 rows)
```

Interpretation:

This query compares customers within the same region, allowing the business to analyze regional customer concentration. It supports region-based marketing and customer behavior analysis.

WINDOW FUNCTION SCREENSHOT

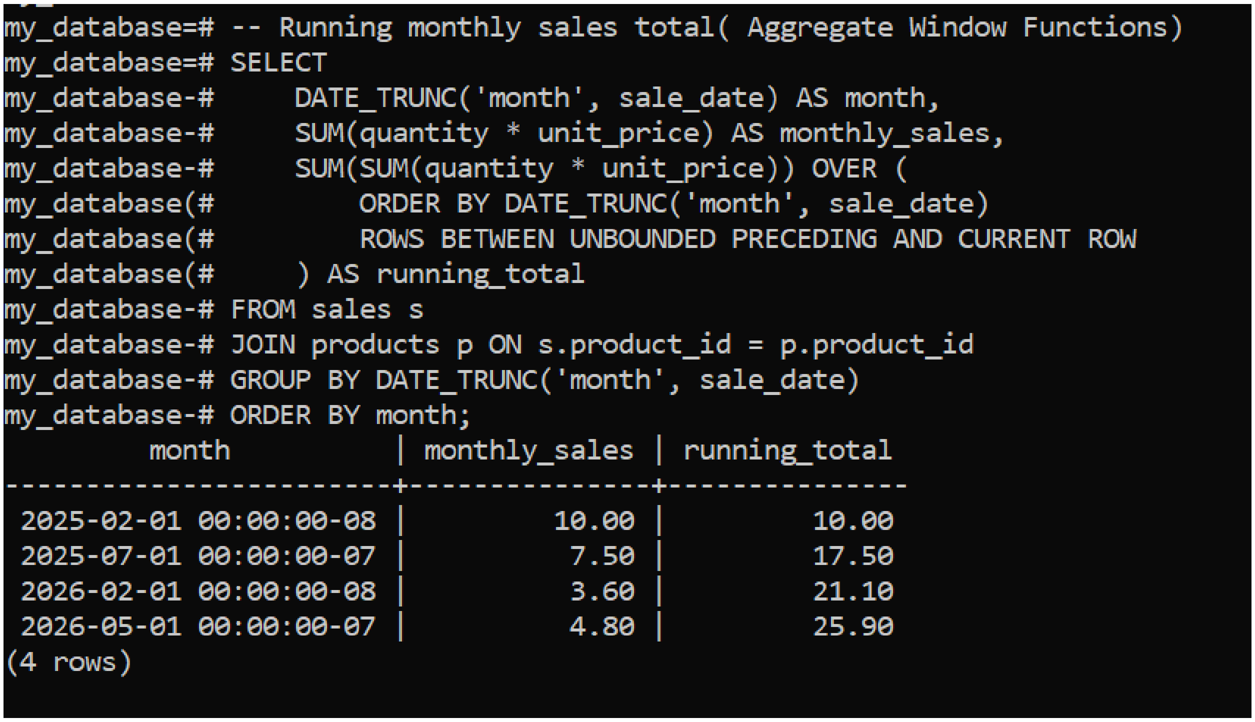
1. Ranking Functions



```
my_database=# -- Rank products by total revenue within each region(RANKING FUNCTION)
my_database=# SELECT
my_database-#     c.region,
my_database-#     p.product_name,
my_database-#     SUM(s.quantity * p.unit_price) AS total_revenue,
my_database-#     RANK() OVER (
my_database(#         PARTITION BY c.region
my_database(#         ORDER BY SUM(s.quantity * p.unit_price) DESC
my_database(#     ) AS product_rank
my_database-# FROM sales s
my_database-# JOIN customers c ON s.customer_id = c.customer_id
my_database-# JOIN products p ON s.product_id = p.product_id
my_database-# GROUP BY c.region, p.product_name;
 region | product_name | total_revenue | product_rank
--------+--------------+---------------+--------------
 East   | Milk         |         13.50 |            1
 East   | Rice         |          4.00 |            2
 West   | Soap         |          4.80 |            1
 West   | Bread        |          3.60 |            2
(4 rows)
```

Interpretation:
This query ranks products based on revenue generated in each region. It enables management to identify the best-performing products and prioritize stocking and promotions accordingly.

2. Aggregate Window Functions (Running Totals)

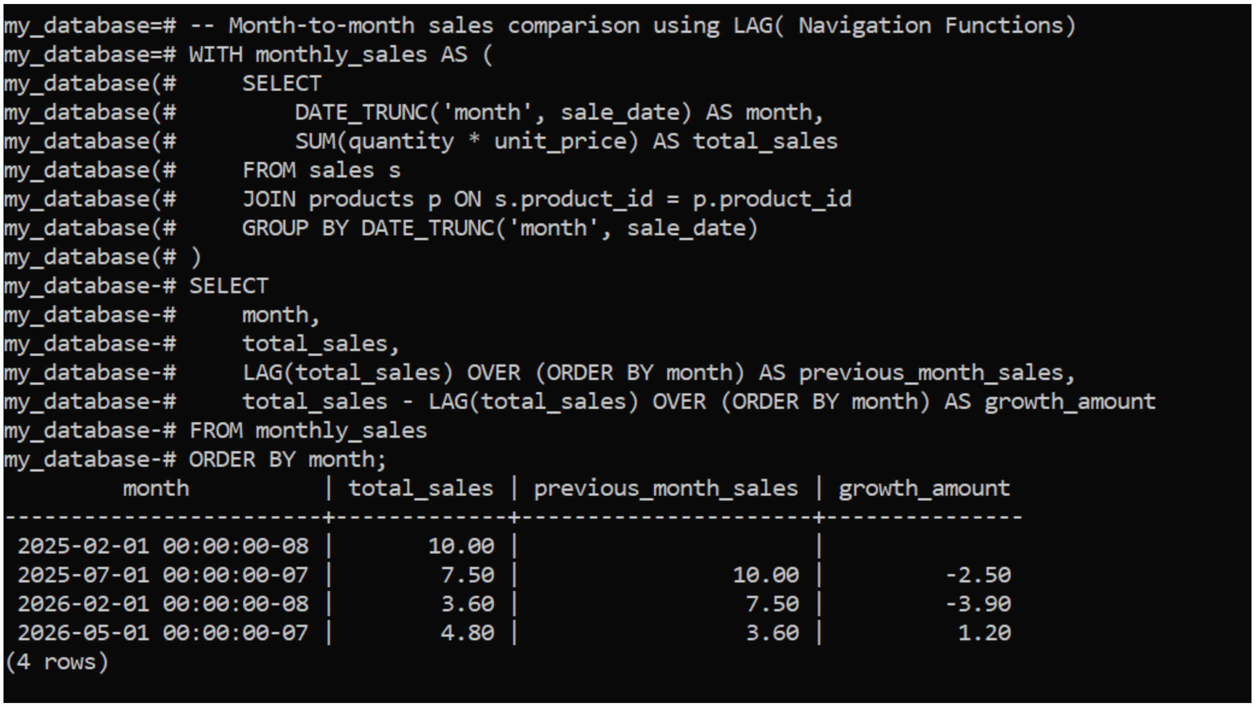


```
my_database=# -- Running monthly sales total( Aggregate Window Functions)
my_database=# SELECT
my_database-#     DATE_TRUNC('month', sale_date) AS month,
my_database-#     SUM(quantity * unit_price) AS monthly_sales,
my_database-#     SUM(SUM(quantity * unit_price)) OVER (
my_database(#         ORDER BY DATE_TRUNC('month', sale_date)
my_database(#         ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
my_database(#     ) AS running_total
my_database-# FROM sales s
my_database-# JOIN products p ON s.product_id = p.product_id
my_database-# GROUP BY DATE_TRUNC('month', sale_date)
my_database-# ORDER BY month;
         month          | monthly_sales | running_total
------------------------+---------------+---------------
 2025-02-01 00:00:00-08 |         10.00 |         10.00
 2025-07-01 00:00:00-07 |          7.50 |         17.50
 2026-02-01 00:00:00-08 |          3.60 |         21.10
 2026-05-01 00:00:00-07 |          4.80 |         25.90
(4 rows)
```

Interpretation:

This query shows how total sales accumulate over time. It helps the business monitor growth trends and evaluate long-term performance.

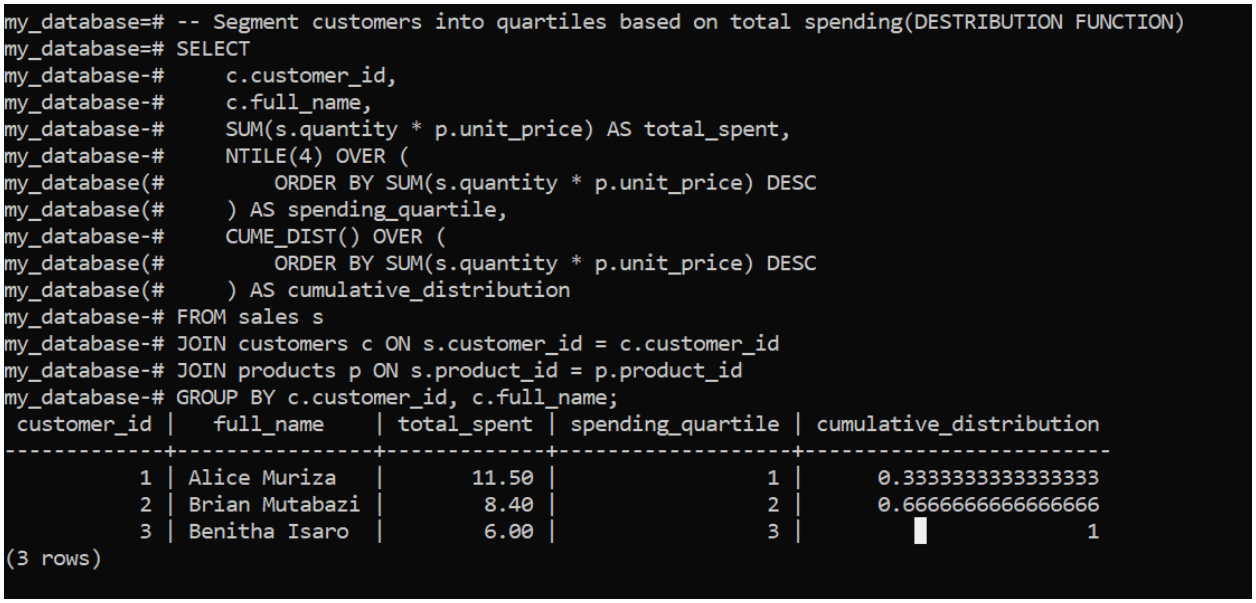
3. Navigation Functions (LAG / LEAD)



```
my_database=# -- Month-to-month sales comparison using LAG( Navigation Functions)
my_database=# WITH monthly_sales AS (
my_database(#     SELECT
my_database(#         DATE_TRUNC('month', sale_date) AS month,
my_database(#         SUM(quantity * unit_price) AS total_sales
my_database(#     FROM sales s
my_database(#     JOIN products p ON s.product_id = p.product_id
my_database(#     GROUP BY DATE_TRUNC('month', sale_date)
my_database(# )
my_database-# SELECT
my_database-#     month,
my_database-#     total_sales,
my_database-#     LAG(total_sales) OVER (ORDER BY month) AS previous_month_sales,
my_database-#     total_sales - LAG(total_sales) OVER (ORDER BY month) AS growth_amount
my_database-# FROM monthly_sales
my_database-# ORDER BY month;
         month          | total_sales | previous_month_sales | growth_amount
------------------------+-------------+----------------------+---------------
 2025-02-01 00:00:00-08 |       10.00 |                      |
 2025-07-01 00:00:00-07 |        7.50 |                10.00 |         -2.50
 2026-02-01 00:00:00-08 |        3.60 |                 7.50 |         -3.90
 2026-05-01 00:00:00-07 |        4.80 |                 3.60 |          1.20
(4 rows)
```

Interpretation:
This query measures sales change between consecutive months. It helps management quickly detect growth or decline and respond with pricing or marketing adjustments.

4. Distribution Functions (NTILE, CUME_DIST)



```
my_database=# -- Segment customers into quartiles based on total spending(DESTRIBUTION FUNCTION)
my_database=# SELECT
my_database-#     c.customer_id,
my_database-#     c.full_name,
my_database-#     SUM(s.quantity * p.unit_price) AS total_spent,
my_database-#     NTILE(4) OVER (
my_database(#         ORDER BY SUM(s.quantity * p.unit_price) DESC
my_database(#     ) AS spending_quartile,
my_database-#     CUME_DIST() OVER (
my_database(#         ORDER BY SUM(s.quantity * p.unit_price) DESC
my_database(#     ) AS cumulative_distribution
my_database-# FROM sales s
my_database-# JOIN customers c ON s.customer_id = c.customer_id
my_database-# JOIN products p ON s.product_id = p.product_id
my_database-# GROUP BY c.customer_id, c.full_name;
 customer_id |   full_name    | total_spent | spending_quartile | cumulative_distribution
-------------+----------------+-------------+-------------------+-------------------------
           1 | Alice Muriza   |       11.50 |                 1 |      0.3333333333333333
           2 | Brian Mutabazi |        8.40 |                 2 |      0.6666666666666666
           3 | Benitha Isaro  |        6.00 |                 3 |                       1
(3 rows)
```

Interpretation:
This query divides customers into four spending groups and shows their cumulative distribution. It supports targeted marketing strategies by identifying high-value and low-value customer segments.