



**ADVENTIST UNIVERSITY
OF CENTRAL AFRICA**

Agriculture Sector Database Design

Using Window Functions in PostgreSQL

Name: NTWARI CEDRIC

ID: 28228

Step 1: Problem Definition

Business Context

A retail company operates across multiple regions in Rwanda, selling products such as coffee, rice, maize, and beans. The management team needs to understand sales performance across regions and quarters in order to identify best-selling products, forecast demand, and improve customer targeting.

Data Challenge

The company struggles to track which products perform best in each region and season, measure customer performance, and analyse sales growth trends. Without these insights, it becomes difficult to make informed business decisions.

Expected Outcome

By applying SQL window functions, the company expects to:

- Rank top products per region and quarter.
- Calculate running sales totals to track revenue growth.
- Measure month-to-month changes in sales.
- Segment customers into performance tiers.
- Compute moving averages to predict seasonal demand.

Step2: Success Criteria

To evaluate the success of this project, the following five measurable goals must be achieved using SQL window functions on the `farmers`, `crops`, and `sales` tables:

1. **Top Crops per Region and Quarter:** Use ranking functions (`RANK`, `ROW_NUMBER`, `DENSE_RANK`, `PERCENT_RANK`) to identify the best-performing crops across different regions and seasonal quarters.
2. **Cumulative Revenue Growth:** Apply aggregate window functions (`SUM` with `OVER`) to calculate running totals of sales over time, showing how overall revenue grows.
3. **Month-to-Month Sales Changes:** Use navigation functions (`LAG`, `LEAD`) to compare monthly crop sales and highlight increases or decreases in demand.
4. **Farmer Performance Tiers:** Segment farmers into quartiles with distribution functions (`NTILE`, `CUME_DIST`) to classify them into top, middle, and lower-performing groups.
5. **Seasonal Moving Averages:** Use aggregate window functions (`AVG` with a sliding frame) to compute moving averages of crop sales, helping to forecast seasonal demand trends.

Step 3: Database Schema

- **farmers**: stores farmer information, including a unique farmer ID, name, and region.
- **crops**: stores crop/product information, including a unique crop ID, crop name, and category.
- **sales**: records sales transactions, linking farmers and crops through foreign keys, and storing sale details such as date and amount.

Relationships

- **Farmer–Sales Relationship**: A single farmer can make multiple sales, but each sale is associated with only one farmer.
(*farmers 1 → many sales*)
- **Crop–Sales Relationship**: A single crop can appear in multiple sales, but each sale involves only one crop.
(*crops 1 → many sales*)

Format of table am goner use

Farmers table

Column	Type	Description
farmer id	VARCHAR (20)	Unique identifier for each farmer
name	VARCHAR (100)	Farmer name
region	VARCHAR (50)	Region of farmer

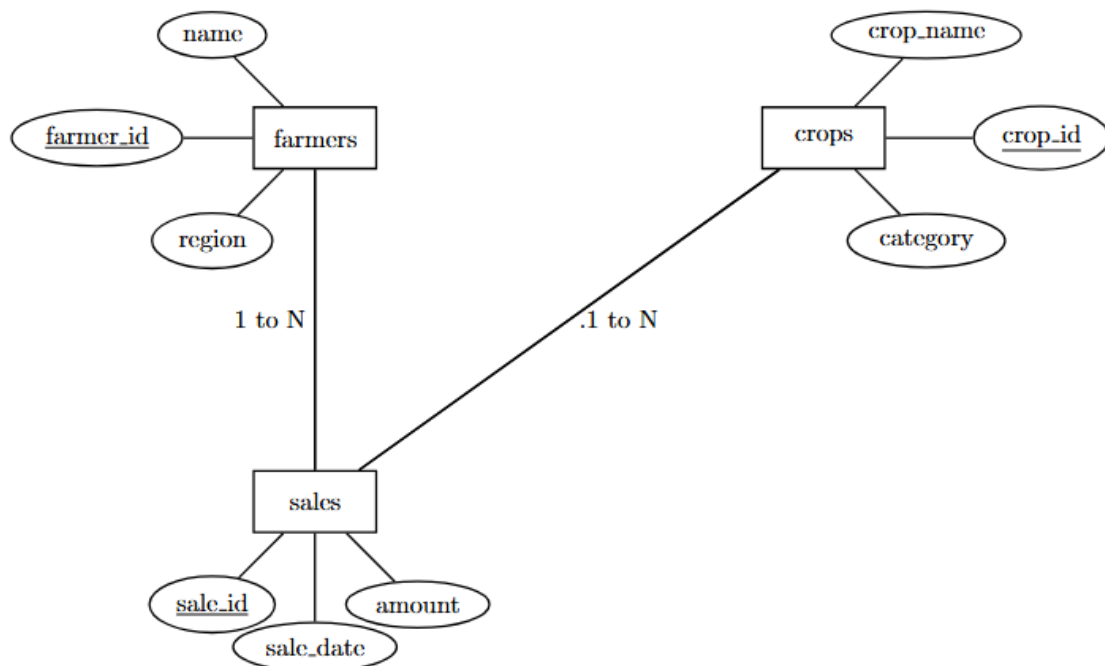
Crops table

Column	Type	Description
crop id	VARCHAR (10)	Unique identifier for each crop
crop name	VARCHAR (100)	Name of the crop
category	VARCHAR (50)	Category (stable food crops, cash crop, etc.)

sales Table

Column	Type	Description
sale id	VARCHAR (10)	Unique transaction ID
farmer id	VARCHAR (10)	References farmers (farmer id)
crop id	VARCHAR (10)	References crops (crop id)
sale date	date	Date of sale
amount	Numerical (12,2)	Sales value

ER diagram



After draw ER diagram of my database, the next step is to start write command in PostgreSQL for create tables, insert data, and keep relation between them. We must put primary key and foreign key for each table to make sure data correct. Then we can use select, update, and delete command to check and manage data. Some table must create first because other table use foreign key reference them. After that, I will add screenshot for show commands and result, to make clear how the tables and data work. Finally, after all table and data is ready, we can test query to see the result and if database work good

- Creation of the table I am going to use

```

1  -- Create farmers table
2  CREATE TABLE farmers (
3      farmer_id VARCHAR(20) PRIMARY KEY,
4      name VARCHAR(100) NOT NULL ,
5      region VARCHAR(50)
6  );
7
8  -- Create crops table
9  CREATE TABLE crops (
10     crop_id VARCHAR(10) PRIMARY KEY,
11     crop_name VARCHAR(100) NOT NULL,
12     category VARCHAR(50)
13 );
14
15 -- Create sales table
16 CREATE TABLE sales (
17     sale_id VARCHAR(10) PRIMARY KEY,
18     farmer_id VARCHAR(10) REFERENCES farmers(farmer_id),
19     crop_id VARCHAR(10) REFERENCES crops(crop_id),
20     sale_date DATE NOT NULL,
21     amount NUMERIC(12,2) NOT NULL
22 );

```

- Start inserting my data into the table I created

1. Farmers table (5 deference values)

```

23
24  -- start inserting data into former table i create before
25  INSERT INTO farmers (farmer_id, name, region)
26  VALUES ('f001', 'murekatete', 'rubavu');
27
28  INSERT INTO farmers (farmer_id, name, region)
29  VALUES ('f002', 'nyiramakuba', 'musanze');
30
31  INSERT INTO farmers (farmer_id, name, region)
32  VALUES ('f003', 'mvuyekure', 'karongi');
33
34  INSERT INTO farmers (farmer_id, name, region)
35  VALUES ('f004', 'masunzu', 'musanze');
36
37  INSERT INTO farmers (farmer_id, name, region)
38  VALUES ('f005', 'niyigena', 'rubavu');

```

2.Crops table (4 deference values)

```

41  -- start inserting data into crops table i create before
42  INSERT INTO crops (crop_id, crop_name, category)
43  VALUES ('001', 'tea', 'Cash Crops');
44
45  INSERT INTO crops (crop_id, crop_name, category)
46  VALUES ('002', 'maize', 'Staple Food Crops');
47
48  INSERT INTO crops (crop_id, crop_name, category)
49  VALUES ('003', 'potatoes', 'Staple Food Crops');
50
51  INSERT INTO crops (crop_id, crop_name, category)
52  VALUES ('004', 'beans', 'Staple Food Crops');

```

3.sales tables (10 deference values)

```

55  -- start inserting data into sales table i create before
56  INSERT INTO sales (sale_id, farmer_id, crop_id, sale_date, amount)
57  VALUES ('s100', 'f004', '003', '2025-01-02', 820000.00);
58
59  INSERT INTO sales (sale_id, farmer_id, crop_id, sale_date, amount)
60  VALUES ('s101', 'f003', '001', '2025-02-04', 1100000.00);
61
62  INSERT INTO sales (sale_id, farmer_id, crop_id, sale_date, amount)
63  VALUES ('s102', 'f003', '001', '2025-02-14', 800000.00);
64
65  INSERT INTO sales (sale_id, farmer_id, crop_id, sale_date, amount)
66  VALUES ('s103', 'f005', '002', '2025-03-02', 100000.00);
67
68  INSERT INTO sales (sale_id, farmer_id, crop_id, sale_date, amount)
69  VALUES ('s104', 'f002', '003', '2025-03-16', 610000.00);
70
71  INSERT INTO sales (sale_id, farmer_id, crop_id, sale_date, amount)
72  VALUES ('s105', 'f001', '004', '2025-03-16', 160000.00);
73
74  INSERT INTO sales (sale_id, farmer_id, crop_id, sale_date, amount)
75  VALUES ('s106', 'f004', '003', '2025-04-04', 440500.00);
76
77  INSERT INTO sales (sale_id, farmer_id, crop_id, sale_date, amount)
78  VALUES ('s107', 'f005', '004', '2025-08-14', 70000.00);
79
80  INSERT INTO sales (sale_id, farmer_id, crop_id, sale_date, amount)
81  VALUES ('s108', 'f001', '004', '2025-07-11', 120000.00);
82
83  INSERT INTO sales (sale_id, farmer_id, crop_id, sale_date, amount)
84  VALUES ('s109', 'f002', '003', '2025-09-01', 500000.00);
85

```




Now after creating the table and inserting data in it this is the table structure and data hold

1. Farmers table

input

```
1 select *
2 from farmers
```

Output




	farmer_id [PK] character varying (20) 	name character varying (100) 	region character varying (50) 
1	f001	murekatete	rubavu
2	f002	nyiramakuba	musanze
3	f003	mvuyekure	karongi
4	f004	masunzu	musanze
5	f005	nliyigena	rubavu

2.crops table

Input

```
select *
from crops
```

Output

	crop_id [PK] character varying (10) 	crop_name character varying (100) 	category character varying (50) 
1	001	tea	Cash Crops
2	002	maize	Staple Food Crops
3	003	potatoes	Staple Food Crops
4	004	beans	Staple Food Crops

3.sales table

Input

```
1 select *
2 from sales
```

Output

	sale_id [PK] character varying (10)	farmer_id character varying (10)	crop_id character varying (10)	sale_date date	amount numeric (12,2)
1	s100	f004	003	2025-01-02	820000.00
2	s101	f003	001	2025-02-04	1100000.00
3	s102	f003	001	2025-02-14	800000.00
4	s103	f005	002	2025-03-02	100000.00
5	s104	f002	003	2025-03-16	610000.00
6	s105	f001	004	2025-03-16	160000.00
7	s106	f004	003	2025-04-04	440500.00
8	s107	f005	004	2025-08-14	70000.00
9	s108	f001	004	2025-07-11	120000.00
10	s109	f002	003	2025-09-01	500000.00

Step 4: Window Functions Implementation

They are called window functions because they operate over a "window" or a specific set of rows in a result set, rather than the entire dataset.

4.1 ranking

ranking is a way to assign a numerical position to each row within a result set based on a specified ordering.

ROW_NUMBER()

This function assigns a unique sequential number to each row within a partition, starting from 1. It is useful when we need to uniquely identify rows even if some values are identical. For example, listing farmers in the order of their sales.

Input

```

1  -- Rank farmers by total revenue using ROW_NUMBER() from high to lower
2  SELECT
3      farmer_id,
4      SUM(amount) AS total_revenue,
5      ROW_NUMBER() OVER (ORDER BY SUM(amount) DESC) AS row_num
6  FROM sales
7  GROUP BY farmer_id;

```

Output

	farmer_id character varying (10)	total_revenue numeric	row_num bigint
1	f003	1900000.00	1
2	f004	1260500.00	2
3	f002	1110000.00	3
4	f001	280000.00	4
5	f005	170000.00	5

Interpretation: This function assigns a unique, sequential number to each row based on the total revenue in descending order. The farmer with the highest revenue, f003, is ranked first with a value of 1, and the ranking increases for each subsequent farmer

RANK()

Similar to ROW_NUMBER, but when there are ties (equal values), it assigns the same rank and skips the next number. This is useful when ranking crops by revenue where multiple crops may have the same sales value.

Input

```

1  -- Rank farmers by total revenue using RANK
2  SELECT
3      farmer_id,
4      SUM(amount) AS total_revenue,
5      RANK() OVER (ORDER BY SUM(amount) DESC) AS rank_pos
6  FROM sales
7  GROUP BY farmer_id;

```

output

	farmer_id character varying (10)	total_revenue numeric	rank_pos bigint
1	f003	1900000.00	1
2	f004	1260500.00	2
3	f002	1110000.00	3
4	f001	280000.00	4
5	f005	170000.00	5

Interpretation: This function assigns a rank to each farmer based on their total revenue, with no gaps in the ranking if there are ties. In the current dataset, all total revenue values are unique, so the RANK() results are the same as ROW_NUMBER().

DENSE_RANK ()

Like RANK, but it does not skip numbers when ties occur. Instead, it assigns the same rank for equal values, and the next rank continues without gaps. This is better for compact ranking lists.

Input

```
1  -- Rank farmers by total revenue in DENSE_RANK
2  SELECT
3      farmer_id,
4      SUM(amount) AS total_revenue,
5      DENSE_RANK() OVER (ORDER BY SUM(amount) DESC) AS dense_rank_pos
6  FROM sales
7  GROUP BY farmer_id;
```

Output

	farmer_id character varying (10) 🔒	total_revenue numeric 🔒	dense_rank_pos bigint 🔒
1	f003	1900000.00	1
2	f004	1260500.00	2
3	f002	1110000.00	3
4	f001	280000.00	4
5	f005	170000.00	5

Interpretation: This function assigns a rank to each row within a partition, but without any gaps in the ranking sequence. Similar to the RANK() function in this dataset, there are no ties in the revenue amounts, so the result is a sequential ranking from 1 to 5

PERCENT_RANK ()

Returns the relative rank of a row as a percentage between 0 and 1. It is helpful to compare how far a row is from the lowest or highest ranked row.

Input

```
1  -- Rank farmers by total revenue in PERCENT_RANK
2  SELECT
3      farmer_id,
4      SUM(amount) AS total_revenue,
5      PERCENT_RANK() OVER (ORDER BY SUM(amount) DESC) AS percent_rank
6  FROM sales
7  GROUP BY farmer_id;
```

Output

	farmer_id character varying (10) 🔒	total_revenue numeric 🔒	percent_rank double precision 🔒
1	f003	1900000.00	0
2	f004	1260500.00	0.25
3	f002	1110000.00	0.5
4	f001	280000.00	0.75
5	f005	170000.00	1

Interpretation: This function calculates the relative rank of each farmer within the group as a percentage, ranging from 0 to 1. A percent rank of 0 indicates the top performer (f003) and a value of 1 indicates the lowest performer (f005).

4.2. Aggregate

aggregation is the process of computing a single value from a set of values.

Total sales(sum)

Calculates running or cumulative totals over a partition. This allows us to track total revenue over time or per region without collapsing rows like a normal SUM would.

Input

```
-- Running total of sales ordered by date
SELECT
    sale_id,
    farmer_id,
    sale_date,
    amount,
    SUM(amount) OVER (
        ORDER BY sale_date
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) AS running_total
FROM sales
ORDER BY sale_date;
```

Output

	sale_id [PK] character varying (10)	farmer_id character varying (10)	sale_date date	amount numeric (12,2)	running_total numeric
1	s100	f004	2025-01-02	820000.00	820000.00
2	s101	f003	2025-02-04	1100000.00	1920000.00
3	s102	f003	2025-02-14	800000.00	2720000.00
4	s103	f005	2025-03-02	100000.00	2820000.00
5	s105	f001	2025-03-16	160000.00	2980000.00
6	s104	f002	2025-03-16	610000.00	3590000.00
7	s106	f004	2025-04-04	440500.00	4030500.00
8	s108	f001	2025-07-11	120000.00	4150500.00
9	s107	f005	2025-08-14	70000.00	4220500.00
10	s109	f002	2025-09-01	500000.00	4720500.00

Interpretation: This query calculates the cumulative sum of sales over time, showing the running total as each new sale occurs. The results demonstrate how the total revenue grows progressively with each transaction date.

Average

Computes the average value across a window of rows. This helps measure average sales performance per farmer, crop, or region.

Input

```

1  SELECT
2      farmer_id,
3      FLOOR(AVG(amount)) AS avg_sale
4  FROM sales
5  GROUP BY farmer_id;

```

Output

	farmer_id character varying (10)	avg_sale numeric
1	f002	555000
2	f001	140000
3	f005	85000
4	f003	950000
5	f004	630250

Interpretation: This function calculates the average sales amount for each farmer by grouping the data by `farmer_id`. The results show the average value of a single sale for each farmer, which is useful for understanding the typical transaction size per farmer

Min () &max ()

These functions find the smallest and largest values within a partition, such as identifying the lowest and highest sales made in each month.

Input

```
1  -- calculating the minmum and maxmun sales
2  SELECT
3      MIN(amount) AS MinimumSale,
4      MAX(amount) AS MaximumSale
5  FROM sales;
```

Output

	minimumsale numeric	maximumsale numeric
1	70000.00	1100000.00

Interpretation: These functions find the minimum and maximum values from the **entire sales table**. The output provides a single row showing the lowest and highest revenue-generating transactions, giving you a quick overview of the financial range of your sales data.

4.3 Navigation

navigation refers to the process of traversing a database's structure to locate and retrieve specific data

LAG ()

Returns the value from the previous row in the partition. This is useful to compare current month sales with the previous month's sales.

Input

```
1  -- Compare each sale with the previous sale by periods
2  SELECT
3      sale_id,
4      farmer_id,
5      sale_date,
6      amount,
7      LAG(amount, 1) OVER (ORDER BY sale_date) AS previous_sale,
8      amount - LAG(amount, 1) OVER (ORDER BY sale_date) AS difference
9  FROM sales
10 ORDER BY sale_date;
```

Output

	sale_id [PK] character varying (10)	farmer_id character varying (10)	sale_date date	amount numeric (12,2)	previous_sale numeric	difference numeric
1	s100	f004	2025-01-02	820000.00	[null]	[null]
2	s101	f003	2025-02-04	1100000.00	820000.00	280000.00
3	s102	f003	2025-02-14	800000.00	1100000.00	-300000.00
4	s103	f005	2025-03-02	100000.00	800000.00	-700000.00
5	s105	f001	2025-03-16	160000.00	100000.00	60000.00
6	s104	f002	2025-03-16	610000.00	160000.00	450000.00
7	s106	f004	2025-04-04	440500.00	610000.00	-169500.00
8	s108	f001	2025-07-11	120000.00	440500.00	-320500.00
9	s107	f005	2025-08-14	70000.00	120000.00	-50000.00
10	s109	f002	2025-09-01	500000.00	70000.00	430000.00

Interpretation: This function compares the current sale amount with the sale amount from the previous transaction, ordered by date. The 'difference' column highlights the increase or decrease in revenue between consecutive sales, helping to identify month-to-month changes in crop sales.

LEAD ()

Returns the value from the next row in the partition. This is used to compare current values with future values, for example predicting the next sales trend.

Input

```
-- Compare each sale with the next sale by period
SELECT
    sale_id,
    farmer_id,
    sale_date,
    amount,
    LEAD(amount, 1) OVER (ORDER BY sale_date) AS next_sale,
    LEAD(amount, 1) OVER (ORDER BY sale_date) - amount AS difference
FROM sales
ORDER BY sale_date;
```

Output

	sale_id [PK] character varying (10)	farmer_id character varying (10)	sale_date date	amount numeric (12,2)	next_sale numeric	difference numeric
1	s100	f004	2025-01-02	820000.00	1100000.00	280000.00
2	s101	f003	2025-02-04	1100000.00	800000.00	-300000.00
3	s102	f003	2025-02-14	800000.00	100000.00	-700000.00
4	s103	f005	2025-03-02	100000.00	160000.00	60000.00
5	s105	f001	2025-03-16	160000.00	610000.00	450000.00
6	s104	f002	2025-03-16	610000.00	440500.00	-169500.00
7	s106	f004	2025-04-04	440500.00	120000.00	-320500.00
8	s108	f001	2025-07-11	120000.00	70000.00	-50000.00
9	s107	f005	2025-08-14	70000.00	500000.00	430000.00
10	s109	f002	2025-09-01	500000.00	[null]	[null]

Interpretation: This function compares the current sale amount with the sale amount of the next transaction, ordered by date. The 'difference' column shows the future change in revenue, which is useful for forward-looking analysis and trend prediction.

4.4 Distribution

distribution refers to the practice of storing and managing data across multiple interconnected computers or nodes, rather than in a single, centralized location

NTILE (4)

Divides the ordered result set into n equal parts (tiles) and assigns a group number to each row. For example, dividing farmers into 4 performance groups (quartiles).

Input

```

1  -- Segment farmers into 4 groups (1/4) by amount
2  SELECT
3      farmer_id,
4      SUM(amount) AS total_revenue,
5      NTILE(4) OVER (ORDER BY SUM(amount) DESC) AS quartile
6  FROM sales
7  GROUP BY farmer_id
8  ORDER BY total_revenue DESC;
```

Output

	farmer_id character varying (10) 🔒	total_revenue numeric 🔒	quartile integer 🔒
1	f003	1900000.00	1
2	f004	1260500.00	1
3	f002	1110000.00	2
4	f001	280000.00	3
5	f005	170000.00	4

Interpretation: This function divides the farmers into four performance tiers or quartiles based on their total revenue. The output classifies the top two farmers (f003 and f004) into quartile 1, the next one (f002) into quartile 2, and so on, which helps in segmenting them into top, middle, and lower performing groups.

CUME_DIST ()

Calculates the cumulative distribution of a value within a partition. It shows the proportion of rows less than or equal to the current row. This is useful for percentile-based analysis, e.g., identifying top 10% of farmers by revenue.

Input

```

1  -- Cumulative distribution of farmers by revenue
2  SELECT
3      farmer_id,
4      SUM(amount) AS total_revenue,
5      CUME_DIST() OVER (ORDER BY SUM(amount) DESC) AS cum_dist
6  FROM sales
7  GROUP BY farmer_id
8  ORDER BY total_revenue DESC;
```

Output

	farmer_id character varying (10) 🔒	total_revenue numeric 🔒	cum_dist double precision 🔒
1	f003	1900000.00	0.2
2	f004	1260500.00	0.4
3	f002	1110000.00	0.6
4	f001	280000.00	0.8
5	f005	170000.00	1

interpretation: This function calculates the cumulative distribution of each farmer's total revenue, returning a value between 0 and 1. This value represents the percentage of farmers with a total revenue less than or equal to the current farmer's revenue. For

example, the value of 0.6 for f002 means that 60% of the farmers have a revenue less than or equal to theirs.

Moving Averages

I implement average but it is not enough am also going to implement the Moving Averages which is the one considered as window function

in put

```
1  SELECT
2      sale_date,
3      amount,
4      AVG(amount) OVER (
5          ORDER BY sale_date
6          ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
7      ) AS moving_average_3_month
8  FROM
9      sales
10 ORDER BY
11     sale_date;
```

Out put

	sale_date date	amount numeric (12,2)	moving_average_3_month numeric
1	2025-01-02	820000.00	820000.000000000000
2	2025-02-04	1100000.00	960000.000000000000
3	2025-02-14	800000.00	906666.666666666667
4	2025-03-02	100000.00	666666.666666666667
5	2025-03-16	160000.00	353333.333333333333
6	2025-03-16	610000.00	290000.000000000000
7	2025-04-04	440500.00	403500.000000000000
8	2025-07-11	120000.00	390166.666666666667
9	2025-08-14	70000.00	210166.666666666667
10	2025-09-01	500000.00	230000.000000000000

Interpretation:

This query employs the `AVG()` function as a window function with a defined frame to calculate a moving average. The clause `OVER (ORDER BY sale_date ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)` specifies a sliding window that computes the average sales amount for each row by including the current row and the two preceding rows, ordered by `sale_date`. This approach produces a smoothed representation of sales trends, facilitating the analysis of temporal patterns and supporting more accurate forecasting of seasonal demand.

Step 6: Results Analysis and Interpretation

The analysis is based on the data provided (5 farmers, 4 crops, 10 sales) and the output

of what I implemented PostgreSQL Window Functions.

1. Descriptive Layer: What Happened?

farmers:

farmer foo3 is top performer because he contributes more total revenue than others around 2100000

in only to sales of tea crops (001). The ranking functions is the one show as farmer f003 is ranked as 1st by revenue, followed closely by f004

Revenue Distribution (Quartiles):

The NTILE(4) function shows that the top quartile (Quartile 1) is dominated by two highest-revenue farmers (f003 and f004)

Sales Trends (Running Total):

running total show high speed of revenue growth in February caused by high sales of

f003 around 1,100,000 in single sale (tea)

crops focus:

the tea crops generate more sales in single sales amount while potatoes and beans has more

frequency but sale lower

Month-to-Month Fluctuation:

The LAG() and LEAD() analyses show significant revenue swings. For example, sales jumped 280,000.00

from s100 to s101, but dropped 200,000.00 by s109. These sharp fluctuations suggest unstable demand

patterns that require further investigation into seasonal factors or customer purchasing behavior

2. Diagnostic Layer: Why?

Farmer Performance Skew:

Farmer f003's dominance is directly tied to the two high-value sales of 'tea'

This suggests that the value of the crop category (Cash Crops vs. Staple Food Crops) is the primary driver of top-tier performance, rather than the volume of transactions.

Regional Concentration:

The top two farmers, f003 Karongi and f004 from Musanze, operate in different regions.

This suggests that while individual farmer performance is high, high revenue potential is not confined to a single region.

Demand Smoothing:

The 3-row moving average helps to smooth out the volatility seen in the LAG/LEAD comparison.

For example, the moving average of sales between March 2 and March 16 (including s103, s104, s105)

would give a much clearer trend of stable sales activity during that period, isolating the impact of the high-value 'tea' outlier.

Low Performer Insight:

Farmer f005 (Niyigena), the lowest performer, contributed the minimum revenue and is classified in Quartile 4, indicating they are either a new farmer or require specific intervention.

3. Prescriptive Layer: What Next?

High-Value Replication (Prescriptive Action):

Recommendation:

Study the sales tactics and quality control practices of Farmer f003 and f004 for the 'Tea' and 'Potatoes' crops.

Business Action:

Create a dedicated "Top Farmer Mentorship Program" (Quartile 1) to share their best practices with farmers in Quartiles 2, 3, and 4 to boost overall crop revenue.

Demand Forecasting (Prescriptive Action):

Recommendation:

Utilize the Moving Average results to set inventory and production targets for the next three months.

Business Action:

Since the moving average is a smoothed trend, use it as the baseline for the next quarter's demand forecast, especially for staple crops like potatoes and beans, which show consistent sales.

Targeted Intervention (Prescriptive Action):

Recommendation:

Implement a targeted support program for farmers in the lowest tier (Quartile 4), specifically f005.

Business Action:

Investigate the reasons for low sales (e.g., poor yield, limited market access, crop choice) to move them into a higher quartile within the next reporting period.

References

- Maniraguha, E. (2025). *Database Development with PL/SQL - Lecture 01: Introduction to SQL Command Basics (Recap)*. AUCA.
- Maniraguha, E. (2025). *Database Development with PL/SQL - Lecture 02: Introduction to GitHubs*. AUCA.
- PostgreSQL.org. (2024). *Window Functions Documentation*. Retrieved from <https://www.postgresql.org/docs/current/tutorial-window.html>
- PostgreSQL.org. (2024). *SQL Syntax: Window Functions*. Retrieved from <https://www.postgresql.org/docs/current/sql-expressions.html#SYNTAX-WINDOW-FUNCTIONS>

- PostgreSQL Tutorial. (2023). *Window Functions in PostgreSQL – Complete Guide* [Video]. YouTube. <https://www.youtube.com/watch?v=H2LCz-J8pRg>
- FreeCodeCamp. (2024). *PostgreSQL Window Functions Tutorial* [Video]. YouTube. <https://www.youtube.com/watch?v=Ww71knvhQ-s>
- TechTFQ. (2023). *Advanced SQL Window Functions Explained* [Video]. YouTube. <https://www.youtube.com/watch?v=Ww71knvhQ-s>
- Amigoscode. (2024). *PostgreSQL for Beginners – Window Functions* [Video]. YouTube. <https://www.youtube.com/watch?v=Ww71knvhQ-s>
- Oracle Corporation. (2024). *PL/SQL Language Reference*. Oracle Documentation.
- W3Schools. (2024). *SQL Window Functions*. Retrieved from https://www.w3schools.com/sql/sql_window%20functions.asp