PostgreSQL

Window Functions & Analytics

Surendra Panpaliya

Agenda

- RANK, DENSE_RANK, ROW_NUMBER
- LAG, LEAD, FIRST_VALUE, SUM() OVER
- Comparison: Oracle analytic functions vs PostgreSQL

Hands-On:

Create product sales ranking report with window functions

Assignment:

Generate per-customer running total using OVER()

Window functions

Provide a way

to perform calculations

across a set of table rows

that are related

to the current row.

Window functions

Useful for performing

Aggregations

Calculations

Over subsets of data

What Are Window Functions?

Perform calculations

across rows that are related

to the current row,

without collapsing results

into groups (unlike GROUP BY).

RANK, DENSE_RANK, and ROW_NUMBER

Function	Purpose
ROW_NUMBER()	Assigns a unique number to each row in the partition
RANK()	Assigns the same rank to tied values, with gaps after the tie
DENSE_RANK()	Assigns the same rank to tied values, but without gaps

WHY are these important in the Telecom Domain?

Telecom datasets are typically large and structured around:

Customer behavior (usage, billing, recharges)

Regional operations (circles, zones)

Temporal sequences (monthly usage, login, recharge)

WHY are these important in the Telecom Domain?

Ranking functions help to:

Identify **top-N customers**

Determine **loyalty tiers**

WHY are these important in the Telecom Domain?

Track	Track first-time actions
Handle	Handle duplicates
Analyze	Analyze subscription/order history

WHERE can they be used?

Use Case	Preferred Function
Top 3 data users per telecom circle	RANK or DENSE_RANK
Loyalty program (Gold, Silver, Bronze)	DENSE_RANK
First recharge of every customer	ROW_NUMBER
Track sequence of plan changes	ROW_NUMBER
Deduplicate transactions	ROW_NUMBER
Rank least active users (churn analysis)	RANK

Comparison Table for Developers

Use Case	Function	Why?
Top N users per circle	RANK()	Allows ties; skips ranks
First transaction/event per group	ROW_NUMBER()	One row per group
Loyalty tiering	DENSE_RANK()	No gaps in ranking
Sequential plan upgrades	ROW_NUMBER()	Ordered tracking
Removing duplicates	ROW_NUMBER()	Keep first; drop rest
Least active customer per circle	RANK()	Useful for churn analysis

Sample Data Setup

```
CREATE TABLE sales (
    sale_id SERIAL PRIMARY KEY,
    customer_name VARCHAR(50),
    sale_amount NUMERIC(10,2),
    city VARCHAR(50)
);
```

Sample Data Setup

```
INSERT INTO sales (customer_name, sale_amount, city) VALUES
('Dev', 500, 'Pune'),
('Dev', 300, 'Pune'),
('Harish', 400, 'Mumbai'),
('Harish', 200, 'Mumbai'),
('Satish', 700, 'Delhi'),
('Satish', 600, 'Delhi');
```

RANK Functions

a) ROW_NUMBER()

Gives a unique row number within a partition.

```
SELECT customer_name, sale_amount, city,
ROW_NUMBER() OVER (PARTITION BY city ORDER BY sale_amount DESC) AS rn
FROM sales;
```

RANK Functions

b) RANK()

Gives rank with gaps (similar to Oracle RANK()).

```
SELECT customer_name, sale_amount, city,
    RANK() OVER (PARTITION BY city ORDER BY sale_amount DESC) AS rnk
FROM sales;
```

RANK Functions

c) DENSE_RANK()

Gives rank without gaps (like Oracle DENSE_RANK()).

```
SELECT customer_name, sale_amount, city,
    DENSE_RANK() OVER (PARTITION BY city ORDER BY sale_amount DESC) AS drnk
FROM sales;
```

RANK vs DENSE_RANK vs ROW_NUMBER

Function	Handles Ties (Same Values)?	Skips Ranks After Ties?	Returns Unique Row Number?	Example Output
RANK()	✅ Yes	✅ Yes	❌ No	1, 2, 2, 4
DENSE_RANK()	✅ Yes	❌ No	❌ No	1, 2, 2, 3
ROW_NUMBER()	❌ No (treats all rows uniquely)	❌ N/A	✅ Yes	1, 2, 3, 4

Why Are These Important?

Use Case	Best Function	Why?
Ranking top performers (with ties)	RANK() or DENSE_RANK()	Keeps fairness for same scores
Deduplicating records	ROW_NUMBER()	Helps in deleting duplicates (keep row number = 1)
Paginating results	ROW_NUMBER()	Easy offset for pages (e.g., 1–10, 11–20, etc.)
Leaderboards	DENSE_RANK()	No gaps in ranks (used in competitions, contests)
Change detection in partitions	All 3	Track order within partitions

Dataset (sales2)

customer_name	sale_amount	city
Dev	500	Pune
Dev	300	Pune
Harish	400	Mumbai
Harish	200	Mumbai
Satish	700	Delhi
Satish	600	Delhi
Satish	700	Delhi
Satish	600	Delhi

Use the SQL Query

```
SELECT
  customer_name,
  sale_amount,
  RANK() OVER (PARTITION BY customer_name ORDER BY sale_amount DESC) AS rank,
  DENSE_RANK() OVER (PARTITION BY customer_name ORDER BY sale_amount DESC) AS dense_rank,
  ROW_NUMBER() OVER (PARTITION BY customer_name ORDER BY sale_amount DESC) AS row_num
FROM sales2;
```

Output of All Three Ranking Functions

customer_name	sale_amount	rank	dense_rank	row_num
Dev	500	1	1	1
Dev	300	2	2	2
Harish	400	1	1	1
Harish	200	2	2	2
Satish	700	1	1	1
Satish	700	1	1	2
Satish	600	3	2	3
Satish	600	3	2	4

🎯 ***RANK()***

- Ties share the same rank.
- Next rank is skipped.
- For Satish:
- Two 700s → **rank 1**,
- Next is **rank 3** (skipping rank 2).

🎯 *DENSE_RANK()*

- Ties share the same rank.
- No rank is skipped.
- For Satish:
- Two 700s → **dense_rank 1**,
- Two 600s → **dense_rank 2**

🎯 *ROW_NUMBER()*

- Assigns **unique row number**, no tie handling.
- Purely orders within each partition.
- For Satish:
- First 700 → 1
- Second 700 → 2
- First 600 → 3
- Second 600 → 4

When to Use What?

Use Case	Recommended Function
Eliminate duplicates or pick top-N rows	ROW_NUMBER()
Assign rankings with gaps for ties	RANK()
Assign rankings with no gaps (dense)	DENSE_RANK()
Show position in contests/leaderboards	RANK() or DENSE_RANK()

LAG() and LEAD()

Window functions

Used to access data from another row

in the result set

without using self-joins.

WHAT are These Functions?

Function	Description
LAG(column)	Returns the previous row’s value
LEAD(column)	Returns the next row’s value
FIRST_VALUE()	Returns the first row’s value in the window
SUM() OVER()	Computes a running or partitioned total (cumulative or grouped total)

LAG() and LEAD()

Function	Purpose
LAG()	Fetches data from a previous row
LEAD()	Fetches data from a next row

Syntax

```
LAG(column_name, offset, default) OVER (PARTITION BY ...
ORDER BY ...)
```

```
LEAD(column_name, offset, default) OVER (PARTITION BY ...
ORDER BY ...)
```

Syntax

column_name: the column whose value you want to fetch.

offset: how many rows behind (LAG) or ahead (LEAD) to look. Default is 1.

default: value to return if the target row doesn’t exist (optional).

Example Table: sales

sale_id	customer_name	sale_amount
1	Dev	500
2	Dev	300
3	Dev	200
4	Harish	400
5	Harish	200

Example 1: Using LAG()

```
SELECT
  customer_name,
  sale_amount,
  LAG(sale_amount) OVER (PARTITION BY customer_name ORDER BY sale_id) AS previous_sale
FROM sales;
```

🧾 Output

customer_name	sale_amount	previous_sale
Dev	500	NULL
Dev	300	500
Dev	200	300
Harish	400	NULL
Harish	200	400

Example 2: Using LEAD()

```
SELECT
  customer_name,
  sale_amount,
  LEAD(sale_amount) OVER (PARTITION BY customer_name ORDER BY sale_id) AS next_sale
FROM sales;
```

🧾 Output

customer_name	sale_amount	next_sale
Dev	500	300
Dev	300	200
Dev	200	NULL
Harish	400	200
Harish	200	NULL

Real-World Use Cases

Use Case	Function Used
Compare current row to previous sales	LAG()
Detect changes in status or value over time	LAG() / LEAD()
Compute differences between rows (delta)	LAG()
Track next appointment or transaction	LEAD()
Detect gaps in time series data	LAG() / LEAD()

Advanced Example: Sales Change Detection

```
SELECT
  customer_name,
  sale_amount,
  sale_amount - LAG(sale_amount) OVER (PARTITION BY customer_name ORDER BY sale_id) AS change
FROM sales;
```

Advanced Example: Sales Change Detection

Use this to identify trends in sales increase or decrease.

Best Practices

- Always use ORDER BY in the OVER() clause to control the sequence.
- Use PARTITION BY when analyzing trends **per customer** or **per group**.
- Use COALESCE(..., 0) if you want to avoid NULLs in your output.

FIRST_VALUE() and LAST_VALUE()

Returns the

first/last value

of a window frame

for each row.

FIRST_VALUE() and LAST_VALUE()

```
SELECT
  customer_name,
  sale_amount,
  FIRST_VALUE(sale_amount) OVER (PARTITION BY customer_name
ORDER BY sale_id) AS first_sale,
  LAST_VALUE(sale_amount) OVER (PARTITION BY customer_name
ORDER BY sale_id ROWS BETWEEN UNBOUNDED PRECEDING
AND UNBOUNDED FOLLOWING) AS last_sale
FROM sales;
```

Note

For LAST_VALUE(),
you **must define the frame** correctly,
else it might return the current row’s value
instead of the true last.

SUM() OVER (...)

- Cumulative or running total
- **per partition or**
- **over all rows.**

SUM() OVER (...)

```
SELECT
  customer_name,
  sale_amount,
  SUM(sale_amount) OVER (PARTITION BY customer_name ORDER BY sale_id) AS running_total
FROM sales;
```

This gives a **progressive sum** for each customer.

AVG() OVER (...)

Returns a **moving average** or
group average
without collapsing rows.

AVG() OVER (...)

```
SELECT
  customer_name,
  sale_amount,
  AVG(sale_amount) OVER (PARTITION BY customer_name) AS avg_sale
FROM sales;
```

Returns the **same average** for each customer group.

COUNT() OVER (...)

Counts the number of rows **in the partition**.

COUNT() OVER (...)

```
SELECT
  customer_name,
  sale_amount,
  COUNT(*) OVER (PARTITION BY customer_name) AS sale_count
FROM sales;
```

NTILE(n)

Breaks ordered data into

n buckets

(quantiles/quartiles).

NTILE(n)

```
SELECT
  customer_name,
  sale_amount,
  NTILE(2) OVER (PARTITION BY customer_name ORDER BY sale_amount DESC) AS quartile
FROM sales;
```

Breaks sales into 2 groups (top/bottom).

Real-Life Use Cases

Use Case	Function
Running total	SUM() OVER
Cumulative average	AVG() OVER
Finding earliest/latest sale per group	FIRST_VALUE() / LAST_VALUE()
Counting rows per group	COUNT() OVER
Percentile distribution (quantiles)	NTILE(n)

Best Practices

Use	Use PARTITION BY for grouped analysis
Use	Use ORDER BY for ordered calculations
Combine	Combine with LAG() or LEAD() to compare current vs. past/future values
Use	Use ROWS BETWEEN carefully for correct frame in cumulative analytics

What Are Analytic Functions?

Oracle	PostgreSQL
Called Analytic Functions	Called Window Functions
Use OVER() clause	Use OVER() clause
Used for ranking, aggregation, previous/next rows	Same

Syntax Comparison

Feature	Oracle SQL	PostgreSQL SQL
Partition	PARTITION BY	PARTITION BY
Ordering	ORDER BY	ORDER BY
Window Frame	ROWS BETWEEN	ROWS BETWEEN
Aggregate over window	SUM() OVER()	SUM() OVER()

Syntax Comparison

Feature	Oracle SQL	PostgreSQL SQL
Rank functions	RANK(), DENSE_RANK(), ROW_NUMBER()	Same
Lag/Lead	LAG(), LEAD()	Same
First/Last	FIRST_VALUE(), LAST_VALUE()	Same
Nth Value	NTH_VALUE()	Same

Function Mapping

Purpose	Oracle	PostgreSQL
Row Number	ROW_NUMBER()	ROW_NUMBER()
Rank	RANK()	RANK()
Dense Rank	DENSE_RANK()	DENSE_RANK()
Running Total	SUM(col) OVER()	SUM(col) OVER()
LAG/LEAD	LAG(), LEAD()	LAG(), LEAD()
First/Last Value	FIRST_VALUE(), LAST_VALUE()	FIRST_VALUE(), LAST_VALUE()
Nth Value	NTH_VALUE()	NTH_VALUE()

Syntax Example: RANK

Oracle

```
SELECT name, salary,
    RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS rnk
FROM employees;
```

Syntax Example: RANK

PostgreSQL

```
SELECT name, salary,
    RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS rnk
FROM employees;
```

Window Frame Defaults

Feature	Oracle Default	PostgreSQL Default
ROWS BETWEEN	UNBOUNDED PRECEDING AND CURRENT ROW (For aggregates)	Same
For RANK, ROW_NUMBER	No frame needed	No frame needed

Partitioning & Ordering

Oracle Example	PostgreSQL Example
PARTITION BY department	PARTITION BY department
ORDER BY salary DESC	ORDER BY salary DESC

Key Differences

Feature	Oracle	PostgreSQL
Functionality	Almost identical	Almost identical
Syntax	Same	Same
Performance	Highly optimized in Oracle	Efficient in PostgreSQL but tuning may differ
Advanced Windows	MATCH_RECOGNIZE (Oracle 12c+)	Not available in core PostgreSQL

Use Case Comparison

Use Case	Oracle	PostgreSQL
Top-N per group	RANK(), ROW_NUMBER()	Same
Running total	SUM() OVER()	Same
Time series gaps	LAG(), LEAD()	Same
Partitioned calculations	PARTITION BY	Same

Summary Table

Analytic Concept	Oracle	PostgreSQL
Rank Functions	✅	✅
Running Totals	✅	✅
LAG/LEAD	✅	✅
Nth Value	✅	✅
Recursion (CONNECT BY)	✅	Use WITH RECURSIVE
Pattern Matching	MATCH_RECOGNIZE	❌ (Use LAG/LEAD + logic)

Conclusion

Overall	Oracle	PostgreSQL
Analytic SQL Coverage	✅ Advanced	✅ Standard SQL (99% compatible)
Syntax	Nearly identical	Nearly identical
Migration Effort	Minimal for window functions	Minimal

Hands-On

- Create a Product Sales Ranking Report using Window Functions in PostgreSQL

Scenario

Task	Window Function
Rank products by sales	RANK()
Handle ties	DENSE_RANK()
Show order of sale entries	ROW_NUMBER()
Calculate total and running total	SUM() OVER()

Assignment

Generate Per-Customer Running Total

Using OVER() in PostgreSQL

Thank you for your support and patience

Surendra Panpaliya

Founder and CEO

GKTCS Innovations

https://www.gktcs.com