SQL WINDOW FUNCTION

(RANKING)

Scenario Based Cheat Sheet

Save this Video for future Reference

Like | Share | Follow

Data Set

Employee	Department	Month	Sales
John	Electronics	Jan	1000
Alice	Electronics	Jan	1200
Bob	Electronics	Feb	900
Clara	Furniture	Jan	800
David	Furniture	Feb	950
Emma	Furniture	Feb	1100

Find each employee's sales along with the **average sales of their department** (for comparison).

Expected Output

Employee	Department	Sales	DeptAvgSales
John	Electronics	1000	1033.33
Alice	Electronics	1200	1033.33
Bob	Electronics	900	1033.33
Clara	Furniture	800	950
David	Furniture	950	950
Emma	Furniture	1100	950

Wrong Approach (Using GROUP BY only):

```
SELECT Department,AVG(Sales) AS AvgSales
FROM Sales
GROUP BY Department;
```

Problem:

- This query gives only one row per department.
- We lose employee-level details.
- Expected output (employee with department avg) is not possible here.

Department	AvgSales
Electronics	1033.33
Furniture	950

Correct Approach Using Window Function

```
SELECT
    Employee,
    Department,
    Sales,
    AVG(Sales) OVER (PARTITION BY Department) AS DeptAvgSales
FROM Sales;
```

Execution Flow:

1. `PARTITION BY Department` → divides rows into groups (Electronics, Furniture).
2. `AVG(Sales)` → average is calculated within each partition.
3. Window function keeps **all rows** and adds computed column.

Definition

A **window function** performs a calculation across a set of rows (called a *window*) that are related to the current row.
Unlike aggregate functions, which collapse rows into a single value, window functions **retain all rows** and add extra calculated columns.

In short:
"They let us calculate running totals, rankings, moving averages, and comparisons without losing row-level detail."

Where We Use Window Functions

Window functions are extremely useful when we need both:

- **Row-level data** (each transaction, each employee, etc.)
- **Aggregated insights** (like ranks, percentages, running totals)

Common Use Cases:

1. Ranking employees based on salary within each department.
2. Calculating cumulative sales by date.
3. Comparing each row with the average value of its group.
4. Finding the first/last order per customer.
5. Calculating moving averages for trend analysis.

Execution Flow of Window Function

1. SQL reads all rows.
2. Rows are grouped into *partitions* (if defined).
3. The window function is applied to each partition **row by row**, without collapsing rows.
4. Each row gets both its own value + aggregated value (like average, rank, sum, etc.).

Types of Window Functions

Window functions can be broadly categorized into four groups:

1. **Aggregate Window Functions**
   - SUM(), AVG(), MIN(), MAX(), COUNT()
   - Example: Running totals, department average, cumulative sales
2. **Ranking Window Functions**
   - ROW_NUMBER(), RANK(), DENSE_RANK(), NTILE()
   - Example: Ranking employees by salary, splitting data into quartiles
3. **Value Window Functions**
   - LEAD(), LAG(), FIRST_VALUE(), LAST_VALUE()
   - Example: Compare today's sales with yesterday's, find first/last order
4. **Analytical Window Functions**
   - PERCENT_RANK(), CUME_DIST(), NTH_VALUE()
   - Example: Finding percentile, cumulative distribution

More Example

Let's consider a **Sales table**:

SaleID	CustomerID	SaleDate	Amount
1	C1	2025-01-01	200
2	C2	2025-01-01	300
3	C1	2025-01-02	150
4	C2	2025-01-02	100
5	C1	2025-01-03	250
6	C3	2025-01-03	400
7	C2	2025-01-04	200
8	C1	2025-01-04	300

Example Where We Didn't Use Window Function (Wrong Approach)

Question:
Find the **running total of sales for each customer** (i.e., cumulative sales over time).

Wrong Query (without window function):

```
SELECT CustomerID, SaleDate, SUM(Amount) AS TotalSales
FROM Sales
GROUP BY CustomerID, SaleDate
ORDER BY CustomerID, SaleDate;
```

Problem:
This query only gives **daily total per customer**, not the **cumulative running total**.

Output (Incorrect):

CustomerID	SaleDate	TotalSales
C1	2025-01-01	200
C1	2025-01-02	150
C1	2025-01-03	250
C1	2025-01-04	300

(We lost the "running total" effect.)

Correct Approach (With Window Function)

```
SELECT
  CustomerID,
  SaleDate,
  Amount,
  SUM(Amount) OVER(PARTITION BY CustomerID ORDER BY SaleDate) AS RunningTotal
FROM Sales
ORDER BY CustomerID, SaleDate;
```

Execution Flow:

1. `PARTITION BY CustomerID` → divide rows by each customer.
2. `ORDER BY SaleDate` → arrange transactions by date within each customer.
3. `SUM() OVER(...)` → compute cumulative sum in order.

Expected Output (Correct):

CustomerID	SaleDate	Amount	RunningTotal
C1	2025-01-01	200	200
C1	2025-01-02	150	350
C1	2025-01-03	250	600
C1	2025-01-04	300	900
C2	2025-01-01	300	300
C2	2025-01-02	100	400
C2	2025-01-04	200	600
C3	2025-01-03	400	400

✔ Now we correctly got the **running total per customer**.

Ranking Customers by Total Purchase

```
SELECT
  CustomerID,
  SUM(Amount) AS TotalPurchase,
  RANK() OVER(ORDER BY SUM(Amount) DESC) AS PurchaseRank
FROM Sales
GROUP BY CustomerID;
```

Output:

CustomerID	TotalPurchase	PurchaseRank
C1	900	1
C2	600	2
C3	400	3

Scenario

Imagine you are a teacher preparing a **result sheet** for a class after a math test.
You want to assign **positions** to students based on their marks.

If we try without **ROW_NUMBER, RANK, DENSE_RANK**:

- Using only GROUP BY → we lose row-level details.
- Using MAX(score) → we only get top score, not tied employees.
- Using subqueries → very complex and unreadable.

👉 So without window functions, it's **either incomplete or messy**.

Here's the table:

StudentName	Marks
A	95
B	90
C	95
D	85
E	90
F	80

---

1. ROW_NUMBER

👉 **Use case:** When you just want a **serial number** (like seat numbers or order of appearance).

- It just gives a **unique number to each row** (no matter if marks are same).
- Think of it as: *"Let's just number them in order."*

```
SELECT
  StudentName, Marks,
  ROW_NUMBER() OVER (ORDER BY Marks DESC) AS RowNum
FROM Students;
```

StudentName	Marks	RowNum
A	95	1
C	95	2
B	90	3
E	90	4
D	85	5
F	80	6

---

2. RANK

👉 **Use case:** When you want to show **positions in competitions** (like 1st, 2nd, 3rd) but **skip ranks** if there are ties.

- Students with the **same marks get the same rank**, but the next rank is **skipped**.
- Think of it as: *"Position with gaps."*

```
SELECT
  StudentName, Marks,
  RANK() OVER (ORDER BY Marks DESC) AS RankPosition
FROM Students;
```

StudentName	Marks	RankPosition
A	95	1
C	95	1
B	90	3
E	90	3
D	85	5
F	80	6

---

4. DENSE_RANK

👉 **Use case:** When you want to show **ranks but without skipping numbers** (like grading systems, employee rating levels, etc.).

- Students with the **same marks get the same rank**, but the next rank is **NOT skipped**.
- Think of it as: *"Position without gaps."*

```
SELECT
  StudentName, Marks,
  DENSE_RANK() OVER (ORDER BY Marks DESC) AS
DenseRankPosition
FROM Students;
```

Output:

StudentName	Marks	DenseRankPosition
A	95	1
C	95	1
B	90	2
E	90	2
D	85	3
F	80	4

Quick Summary (Easiest Way):

- **ROW_NUMBER** → Just numbering, no matter if values are same.
- **RANK** → Gives same rank for ties but skips the next numbers.
- **DENSE_RANK** → Gives same rank for ties but does not skip numbers.