Introduction to SQL Window Functions

What are Window Functions?

- Window functions perform calculations across a set of rows related to the current row.
- Imagine a "window" sliding over your data. For each row, the function "looks" at the rows within that window and calculates something.
- Key difference from GROUP BY: Window functions do not collapse rows. A result is returned for each row.
- They allow you to keep the detail of your data while adding analytical information to it.

Key Benefits of Using Window Functions

- **Ranking**: Numbering or assigning ranks (top 5 products, employee ratings).
- **Running Calculations:** Calculating cumulative sums/averages (month-to-date sales, moving average).
- **Row-to-Row Comparisons:** Accessing data from previous/next rows (difference from the previous day, % change).
- **Group-Wise Analysis:** Performing the above operations independently within different categories (ranking products within each category).

How to Write: General Syntax

```
SELECT
    column1,
    column2,
    FUNCTION_NAME(arguments) OVER (
        [PARTITION BY partition_expression]
        [ORDER BY sort_expression]
        [frame_clause] -- Window frame (optional)
    ) AS result_name
FROM
    your_table;
```

- Let's look at each part inside OVER() in detail.

Inside OVER(): Defining the Window

- The OVER() clause specifies how the window function operates. It uses sub-clauses:

```
FUNCTION_NAME(...) OVER (
    [PARTITION BY ...] -- Defines groups
    [ORDER BY ...]     -- Defines order within groups
    [frame_clause]   -- Defines row subset (later)
)
```

- FUNCTION_NAME(...): The window function being applied (e.g., ROW_NUMBER(), SUM()).
- PARTITION BY partition_expression (Optional):
  - Divides rows into independent partitions (groups).
  - Calculations restart for each partition.
  - If omitted, the entire dataset is one partition.
- ORDER BY sort_expression (Optional, but often crucial):
  - Sorts rows within each partition.
  - Essential for sequence-dependent functions (ROW_NUMBER, RANK, LAG, LEAD, running totals).
  - Defines "previous"/"next" row concepts.

Function: ROW_NUMBER()

- **Purpose:** Assigns a unique sequential number to each row within its partition, according to the ORDER BY clause.
- **Result:** Always 1, 2, 3, 4... without gaps, even if values are tied.
- **Use Case:** Simple row numbering within a group.

```sql
-- Number employees by salary within each department
SELECT
    employee_name,
    department_id,
    salary,
    ROW_NUMBER() OVER (PARTITION BY department_id ORDER BY salary DESC) as rn
FROM employees;
```

employee_name	department_id	salary	rn
Alice	10	90000	1
Bob	10	85000	2
Charlie	10	78000	3
David	20	120000	1
Eve	20	115000	2
Fiona	20	115000	3
George	20	100000	4
Hannah	30	75000	1
Ian	30	72000	2

Function: RANK()

- **Purpose:** Assigns a rank to each row within its partition based on the ORDER BY clause.
- **Tie Handling:** Rows with the same value (per ORDER BY) receive the same rank. The next rank is skipped.
- **Result:** 1, 2, 2, 4, 5... (rank 3 is skipped).
- **Use Case:** Determining positions in a ranking where ties are possible and gaps are acceptable.

```sql
-- Rank employees by salary (with gaps for ties)
SELECT
    employee_name, department_id, salary,
    RANK() OVER (PARTITION BY department_id ORDER BY salary DESC) as salary_rank
FROM employees;
```

employee_name	department_id	salary	salary_rank
Alice	10	90000	1
Bob	10	85000	2
Carol	10	85000	2
Dave	10	70000	4
Eve	20	120000	1
Frank	20	110000	2
Grace	20	110000	2
Heidi	20	110000	2
Ian	20	105000	5

Function: DENSE_RANK()

- **Purpose:** Similar to RANK(), but assigns ranks without gaps.
- **Tie Handling:** Rows with the same value receive the same rank, but the next rank immediately follows.
- **Result:** 1, 2, 2, 3, 4... (rank 3 is not skipped).
- **Use Case:** Dense ranking without gaps.

```sql
-- Dense rank employees by salary (no gaps for ties)
SELECT
    employee_name, department_id, salary,
    DENSE_RANK() OVER (PARTITION BY department_id ORDER BY salary DESC) as dense_salary_rank
FROM employees;
```

employee_name	department_id	salary	dense_salary_rank
Alice	10	90000	1
Bob	10	85000	2
Carol	10	85000	2
Dave	10	70000	3
Eve	20	120000	1
Frank	20	110000	2
Grace	20	110000	2
Heidi	20	110000	2
Ian	20	105000	3

Function: LAG()

- **Purpose:** Accesses data from a previous row within the partition, according to the ORDER BY sequence.
- **Parameters:** LAG(expression, [offset], [default_value])
  - **offset:** How many rows back to look (default is 1).
  - **default_value:** What to return if there is no preceding row (e.g., for the first row).
- **Use Case:** Comparing the current value with the previous one (e.g., today's sales vs. yesterday's sales).

```sql
-- Compare sales with the previous month
SELECT
    sale_month, monthly_sales,
    LAG(monthly_sales, 1, 0) OVER (ORDER BY sale_month) as previous_month_sales
FROM monthly_sales_summary;
```

sale_month	monthly_sales	previous_month_sales
2025-01	10000	0
2025-02	12000	10000
2025-03	11000	12000
2025-04	13000	11000

Function: LEAD()

- **Purpose:** Accesses data from a subsequent row within the partition, according to the ORDER BY sequence.
- **Parameters:** LEAD(expression, [offset], [default_value]) (similar to LAG).
- **Use Case:** Forecasting, comparing with the next period.

```sql
-- Compare sales with the next month
SELECT
    sale_month, monthly_sales,
    LEAD(monthly_sales, 1, 0) OVER (ORDER BY sale_month) as next_month_sales
FROM monthly_sales_summary;
```

sale_month	monthly_sales	next_month_sales
2025-01	10000	12000
2025-02	12000	11000
2025-03	11000	13000
2025-04	13000	0

Function: NTILE(n)

- **Purpose:** Divides the rows within each partition into n roughly equal groups (buckets).
- **Result:** Returns the bucket number (from 1 to n) for each row.
- **Use Case:** Dividing customers into quartiles based on purchase volume, identifying the top 10% of products.

```sql
-- Divide products into 5 price categories
SELECT
    product_name, price,
    NTILE(5) OVER (ORDER BY price) as price_category
FROM products;
```

product_name	price	price_category
P1	10	1
P2	20	1
P3	25	2
P4	30	2
P5	40	3
P6	50	3
P7	60	4
P8	75	4
P9	90	5
P10	100	5

Window Frame: Precise Control

- This optional part of OVER() defines the exact subset of rows (the frame) within the partition used for the function's calculation for the current row.
- Especially useful with **aggregate functions (SUM, AVG, COUNT, MIN, MAX)** used as window functions.
- **Syntax:** ROWS | RANGE BETWEEN <start> AND <end>
- <start> **and** <end> **can be**:
  - UNBOUNDED PRECEDING (start of partition)
  - n PRECEDING (n rows before current)
  - CURRENT ROW (the current row)
  - n FOLLOWING (n rows after current)
  - UNBOUNDED FOLLOWING (end of partition)

Example: Running Total

- **ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW:** For each row, take all rows from the start of the partition (here, the whole table as there's no **PARTITION BY**) up to and including the current row, and calculate their sum.

```sql
SELECT
    order_date,
    daily_amount,
    SUM(daily_amount) OVER (
        ORDER BY order_date
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) as running_total
FROM daily_orders;
```

order_date	daily_amount	running_total
2025-04-15	200	200
2025-04-16	300	500
2025-04-17	150	650
2025-04-18	250	900

Thank you

- Author: Denys Kuchmei
- My LinkedIn: https://www.linkedin.com/in/denys-kuchmei-7a8259208/
- Date: April 2025
- Join Codeus community in Discord
- Join Codeus community in LinkedIn