Raushan Kumar

# WINDOW FUNCTION

1. ROW_NUMBER()
2. RANK()
3. DENSE_RANK()
4. NTILE()
5. LEAD()
6. LAG()
7. FIRST_VALUE()
8. LAST_VALUE()
9. CUME_DIST()
10. PERCENT_RANK()
11. SUM()
12. AVG()



Raushan Kumar
https://www.linkedin.com/in/raushan-kumar-553154297/


**Window Functions in SQL** allow you to perform operations across a set of table rows that are related to the current row. These functions are useful for performing calculations like running totals, ranking, or cumulative sums without the need for self-joins or subqueries. A window function operates over a "window" of rows defined by the OVER clause.

**Key Concepts:**

- **Window**: The set of rows over which the window function is applied. This is defined using the OVER clause.
- **Partitioning**: Dividing the result set into groups. You can partition data by one or more columns.
- **Ordering**: Specifies the order of the rows in each partition.
- **Frame**: Specifies a subset of rows within a partition to apply the window function to.

# Common Window Functions in SQL

## 1. ROW_NUMBER()

Assigns a unique number to each row within a partition, starting from 1. It is useful for ranking rows.

**Syntax:**

ROW_NUMBER() OVER (PARTITION BY column1 ORDER BY column2)

**Example:**

Let's say we have a table employee with

**Columns:**

- employee_id
- employee_name
- department
- salary

**Create employee Table**

```
CREATE TABLE employee (
   employee_id INT PRIMARY KEY,
   employee_name VARCHAR(100),
   department VARCHAR(100),
   salary DECIMAL(10, 2)
);
```

## Insert records in the Employee table

```
INSERT INTO employee (employee_id, employee_name, department, salary)
VALUES
(1, 'Alice', 'HR', 50000.00),
(2, 'Bob', 'Finance', 60000.00),
(3, 'Charlie', 'Finance', 55000.00),
(4, 'David', 'HR', 45000.00),
(5, 'Eve', 'IT', 70000.00),
(6, 'Frank', 'IT', 65000.00),
(7, 'Grace', 'HR', 50000.00),
(8, 'Heidi', 'Finance', 75000.00),
(9, 'Ivan', 'IT', 72000.00),
(10, 'Judy', 'HR', 51000.00);
```

## Display the Data

```
SELECT * FROM employee;
```

| employee_id | employee_name | department | salary |
|---|---|---|---|
| 1 | Alice | HR | 50000 |
| 2 | Bob | Finance | 60000 |
| 3 | Charlie | Finance | 55000 |
| 4 | David | HR | 45000 |
| 5 | Eve | IT | 70000 |
| 6 | Frank | IT | 65000 |
| 7 | Grace | HR | 50000 |
| 8 | Heidi | Finance | 75000 |
| 9 | Ivan | IT | 72000 |
| 10 | Judy | HR | 51000 |

## ROW_NUMBER()

SELECT employee_id,employee_name,department,salary,

ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) AS rowNumber FROM employee;

| employee_id | employee_name | department | salary | rowNumber |
|---|---|---|---|---|
| 8 | Heidi | Finance | 75000 | 1 |
| 2 | Bob | Finance | 60000 | 2 |
| 3 | Charlie | Finance | 55000 | 3 |
| 10 | Judy | HR | 51000 | 1 |
| 1 | Alice | HR | 50000 | 2 |
| 7 | Grace | HR | 50000 | 3 |
| 4 | David | HR | 45000 | 4 |
| 9 | Ivan | IT | 72000 | 1 |
| 5 | Eve | IT | 70000 | 2 |
| 6 | Frank | IT | 65000 | 3 |

**Explanation:**

- This will partition the employee based on “department” and rank employees based on their “salary” with the highest salary receiving row number 1.
- The ORDER BY clause specifies how to order the rows within the window.

## 2. RANK()

Similar to ROW_NUMBER(), but if two rows have the same value in the ORDER BY clause, they receive the same rank. The next rank is skipped.

**Syntax:**

RANK() OVER (PARTITION BY column1 ORDER BY column2)

### RANK()

SELECT employee_id,employee_name,department,salary, RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS salary_rank FROM employee;

| employee_id | employee_name | department | salary | salary_rank |
|---|---|---|---|---|
| 8 | Heidi | Finance | 75000 | 1 |
| 2 | Bob | Finance | 60000 | 2 |
| 3 | Charlie | Finance | 55000 | 3 |
| 10 | Judy | HR | 51000 | 1 |
| **1** | **Alice** | **HR** | **50000** | 2 |
| **7** | **Grace** | **HR** | **50000** | 2 |
| 4 | David | HR | 45000 | 4 |
| 9 | Ivan | IT | 72000 | 1 |
| 5 | Eve | IT | 70000 | 2 |
| 6 | Frank | IT | 65000 | 3 |

**Explanation:**

- Employees with the same salary will receive the same rank.
- The next rank will be skipped (e.g., if two employees share rank 2, the next rank will be 4).

## 3. DENSE_RANK()

Works like RANK(), but without skipping ranks if there are ties.

**Syntax:**

DENSE_RANK() OVER (PARTITION BY column1 ORDER BY column2)

### DENSE_RANK()

SELECT employee_id,employee_name,department,salary, DENSE_RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS salary_dense_rank FROM employee;

| employee_id | employee_name | department | salary | salary_dense_rank |
|---|---|---|---|---|
| 8 | Heidi | Finance | 75000 | 1 |
| 2 | Bob | Finance | 60000 | 2 |
| 3 | Charlie | Finance | 55000 | 3 |
| 10 | Judy | HR | 51000 | 1 |
| 1 | Alice | HR | 50000 | 2 |
| 7 | Grace | HR | 50000 | 2 |
| 4 | David | HR | 45000 | 3 |
| 9 | Ivan | IT | 72000 | 1 |
| 5 | Eve | IT | 70000 | 2 |
| 6 | Frank | IT | 65000 | 3 |

**Explanation:**

- Employees with the same salary will receive the same rank.
- The next rank will not be skipped (e.g., if two employees share rank 2, the next rank will be 3).

## 4. NTILE(n)

The NTILE() window function in SQL divides the result set into a specified number of roughly equal parts or "buckets" and assigns a unique bucket number to each row in the result set. This function is useful for tasks like distributing data into quantiles, percentiles, or any other groupings that require evenly splitting the rows.

**Syntax:**

NTILE(n) OVER (PARTITION BY column1 ORDER BY column2)

**NTILE(n)**

SELECT employee_id,employee_name,department,salary, NTILE(4) OVER (PARTITION BY department ORDER BY salary DESC) AS quartile FROM employee;

| employee_id | employee_name | department | salary | quartile |
|---|---|---|---|---|
| 8 | Heidi | Finance | 75000 | 1 |
| 2 | Bob | Finance | 60000 | 2 |
| 3 | Charlie | Finance | 55000 | 3 |
| 10 | Judy | HR | 51000 | 1 |
| 1 | Alice | HR | 50000 | 2 |
| 7 | Grace | HR | 50000 | 3 |
| 4 | David | HR | 45000 | 4 |
| 9 | Ivan | IT | 72000 | 1 |
| 5 | Eve | IT | 70000 | 2 |
| 6 | Frank | IT | 65000 | 3 |

**Explanation:**

- This divides the employees into 4 groups (quartiles), assigning each employee to a quartile based on their salary.

- Employees in the highest quartile will have a quartile value of 1, the second highest will have 2, etc.

## 5. LEAD()

Returns the value of a specified expression for the row that follows the current row within the same result set.

**Syntax:**

LEAD(column_name, n, default) OVER (PARTITION BY column1 ORDER BY column2)

**LEAD()**

SELECT employee_id,employee_name,department,salary, LEAD(salary,1,0) OVER (PARTITION BY department ORDER BY salary DESC) AS next_row_salary FROM employee;

| employee_id | employee_name | department | salary | next_row_salary |
|---|---|---|---|---|
| 8 | Heidi | Finance | 75000 | 60000 |
| 2 | Bob | Finance | 60000 | 55000 |
| 3 | Charlie | Finance | 55000 | 0 |
| 10 | Judy | HR | 51000 | 50000 |
| 1 | Alice | HR | 50000 | 50000 |
| 7 | Grace | HR | 50000 | 45000 |
| 4 | David | HR | 45000 | 0 |
| 9 | Ivan | IT | 72000 | 70000 |
| 5 | Eve | IT | 70000 | 65000 |
| 6 | Frank | IT | 65000 | 0 |

**Explanation:**

- LEAD(salary, 1,0) returns the salary of the employee in the next row.
- If there is no next row, it returns the default value (0 in this case).

## 6. LAG()

Returns the value of a specified expression for the row that precedes the current row within the same result set.

**Syntax:**

LAG(column_name, n, default) OVER (PARTITION BY column1 ORDER BY column2)

**LAG()**

SELECT employee_id,employee_name,department,salary, LAG(salary,1,0) OVER (PARTITION BY department ORDER BY salary DESC) AS previous_row_salary FROM employee;

| employee_id | employee_name | department | salary | previous_row_salary |
|---|---|---|---|---|
| 8 | Heidi | Finance | 75000 | 0 |
| 2 | Bob | Finance | 60000 | 75000 |
| 3 | Charlie | Finance | 55000 | 60000 |
| 10 | Judy | HR | 51000 | 0 |
| 1 | Alice | HR | 50000 | 51000 |
| 7 | Grace | HR | 50000 | 50000 |
| 4 | David | HR | 45000 | 50000 |
| 9 | Ivan | IT | 72000 | 0 |
| 5 | Eve | IT | 70000 | 72000 |
| 6 | Frank | IT | 65000 | 70000 |

**Explanation:**

- LAG(salary, 1,0) returns the salary of the employee in the previous row.
- If there is no previous row, it returns the default value (0 in this case).

## 7. FIRST_VALUE()

Returns the first value in an ordered set of rows.

**Syntax:**

FIRST_VALUE(column_name) OVER (PARTITION BY column1 ORDER BY column2)

### FIRST_VALUE()

SELECT employee_id,employee_name,department,salary, FIRST_VALUE(salary) OVER (PARTITION BY department ORDER BY salary DESC) AS highest_salary FROM employee;

| employee_id | employee_name | department | salary | highest_salary |
|---|---|---|---|---|
| 8 | Heidi | Finance | 75000 | 75000 |
| 2 | Bob | Finance | 60000 | 75000 |
| 3 | Charlie | Finance | 55000 | 75000 |
| 10 | Judy | HR | 51000 | 51000 |
| 1 | Alice | HR | 50000 | 51000 |
| 7 | Grace | HR | 50000 | 51000 |
| 4 | David | HR | 45000 | 51000 |
| 9 | Ivan | IT | 72000 | 72000 |
| 5 | Eve | IT | 70000 | 72000 |
| 6 | Frank | IT | 65000 | 72000 |

**Explanation:**

- This will return the highest salary for all employees in the result set, because the first value is determined by ordering the data by salary in descending order.

## 8. LAST_VALUE()

Returns the last value in an ordered set of rows.

**Syntax:**

LAST_VALUE(column_name) OVER (PARTITION BY column1 ORDER BY column2 ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)

### LAST_VALUE()

SELECT employee_id,employee_name,department,salary, LAST_VALUE(salary) OVER (PARTITION BY department ORDER BY salary DESC ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)

AS lowest_salary FROM employee;

| employee_id | employee_name | department | salary | lowest_salary |
|---|---|---|---|---|
| 8 | Heidi | Finance | 75000 | 55000 |
| 2 | Bob | Finance | 60000 | 55000 |
| 3 | Charlie | Finance | 55000 | 55000 |
| 10 | Judy | HR | 51000 | 45000 |
| 1 | Alice | HR | 50000 | 45000 |
| 7 | Grace | HR | 50000 | 45000 |
| 4 | David | HR | 45000 | 45000 |
| 9 | Ivan | IT | 72000 | 65000 |
| 5 | Eve | IT | 70000 | 65000 |
| 6 | Frank | IT | 65000 | 65000 |

**Explanation:**

- This will return the lowest salary for all employees in the result set.

## 9. CUME_DIST()

Calculates the cumulative distribution of a value in a group of rows. This function returns the relative position of a row in a sorted set of rows.

**Syntax:**

CUME_DIST() OVER (PARTITION BY column1 ORDER BY column2)

### CUME_DIST()

SELECT employee_id,employee_name,department,salary, CUME_DIST() OVER (PARTITION BY department ORDER BY salary DESC ) AS cum_dist FROM employee;

| employee_id | employee_name | department | salary | cum_dist |
|---|---|---|---|---|
| 8 | Heidi | Finance | 75000 | 0.333333333 |
| 2 | Bob | Finance | 60000 | 0.666666667 |
| 3 | Charlie | Finance | 55000 | 1 |
| 10 | Judy | HR | 51000 | 0.25 |
| 1 | Alice | HR | 50000 | 0.75 |
| 7 | Grace | HR | 50000 | 0.75 |
| 4 | David | HR | 45000 | 1 |
| 9 | Ivan | IT | 72000 | 0.333333333 |
| 5 | Eve | IT | 70000 | 0.666666667 |
| 6 | Frank | IT | 65000 | 1 |

**Explanation:**

- The CUME_DIST function returns a value between 0 and 1, indicating the relative position of the current row in the ordered result set.

## 10. PERCENT_RANK()

Calculates the relative rank of a row within a group of rows, expressed as a percentage between 0 and 1.

**Syntax:**

PERCENT_RANK() OVER (PARTITION BY column1 ORDER BY column2)

### PERCENT_RANK()

SELECT employee_id,employee_name,department,salary, PERCENT_RANK() OVER (PARTITION BY department ORDER BY salary DESC ) AS prcnt_rank FROM employee;

| employee_id | employee_name | department | salary | prcnt_rank |
|---|---|---|---|---|
| 8 | Heidi | Finance | 75000 | 0 |
| 2 | Bob | Finance | 60000 | 0.5 |
| 3 | Charlie | Finance | 55000 | 1 |
| 10 | Judy | HR | 51000 | 0 |
| 1 | Alice | HR | 50000 | 0.333333333 |
| 7 | Grace | HR | 50000 | 0.333333333 |
| 4 | David | HR | 45000 | 1 |
| 9 | Ivan | IT | 72000 | 0 |
| 5 | Eve | IT | 70000 | 0.5 |
| 6 | Frank | IT | 65000 | 1 |

**Explanation:**

- The PERCENT_RANK function returns the rank of each row as a percentage of the total number of rows in the partition.

## 11. SUM()

This window function computes the running total or cumulative sum of values over a set of rows.

**Syntax:**

SUM(column_name) OVER (PARTITION BY column1 ORDER BY column2)

**SUM()**

SELECT employee_id,employee_name,department,salary, SUM(salary) OVER (PARTITION BY department ORDER BY salary DESC ) AS running_total FROM employee;

| employee_id | employee_name | department | salary | running_total |
|---|---|---|---|---|
| 8 | Heidi | Finance | 75000 | 75000 |
| 2 | Bob | Finance | 60000 | 135000 |
| 3 | Charlie | Finance | 55000 | 190000 |
| 10 | Judy | HR | 51000 | 51000 |
| 1 | Alice | HR | 50000 | 151000 |
| 7 | Grace | HR | 50000 | 151000 |
| 4 | David | HR | 45000 | 196000 |
| 9 | Ivan | IT | 72000 | 72000 |
| 5 | Eve | IT | 70000 | 142000 |
| 6 | Frank | IT | 65000 | 207000 |

**Explanation:**

- This calculates the cumulative sum of salaries in ascending order.

## 12. AVG()

Calculates the running average of a column over a window of rows.

**Syntax:**

AVG(column_name) OVER (PARTITION BY column1 ORDER BY column2)

**AVG()**

SELECT employee_id,employee_name,department,salary, AVG(salary) OVER (PARTITION BY department ORDER BY salary DESC ) AS running_avg FROM employee;

| employee_id | employee_name | department | salary | running_avg |
|---|---|---|---|---|
| 8 | Heidi | Finance | 75000 | 75000 |
| 2 | Bob | Finance | 60000 | 67500 |
| 3 | Charlie | Finance | 55000 | 63333.33333 |
| 10 | Judy | HR | 51000 | 51000 |
| 1 | Alice | HR | 50000 | 50333.33333 |
| 7 | Grace | HR | 50000 | 50333.33333 |
| 4 | David | HR | 45000 | 49000 |
| 9 | Ivan | IT | 72000 | 72000 |
| 5 | Eve | IT | 70000 | 71000 |
| 6 | Frank | IT | 65000 | 69000 |

**Explanation:**

- This calculates the running average of salaries in ascending order.

## Conclusion

Window functions are incredibly powerful for performing various types of calculations on data, such as running totals, ranking, and comparisons. The key to using window functions effectively is understanding the OVER clause, which allows you to specify how the window is defined in terms of ordering and partitioning.

- **ROW_NUMBER(), RANK(), DENSE_RANK()**: Ranking functions.
- **LEAD(), LAG()**: Accessing data from other rows in the result set.
- **SUM(), AVG(), FIRST_VALUE(), LAST_VALUE()**: Aggregation and cumulative operations.
- **CUME_DIST(), PERCENT_RANK()**: Distribution and ranking calculations.



By: Raushan Kumar

Please follow for more such content:

https://www.linkedin.com/in/raushan-kumar-553154297/