### What are Window Functions?

Window functions perform calculations across a set of table rows related to the current row within a **window**, defined by the OVER() clause. Unlike aggregate functions (e.g., SUM, AVG), window functions do not collapse rows into a single result; instead, they return a value for each row.

### Key Features of Window Functions:

1. Operate on a subset of rows called a **window**.
2. Allow calculations across rows without grouping the entire dataset.
3. Useful for ranking, running totals, moving averages, and more.

### Scenarios to Use Window Functions:

1. **Ranking rows**: Assign ranks based on a column's value.
2. **Aggregating without grouping**: Compute aggregates like SUM or AVG without collapsing rows.
3. **Comparisons within a partition**: Compare rows within subsets (e.g., top performers by department).
4. **Running totals or cumulative sums**: Compute totals incrementally across rows.
5. **Lagging/leading values**: Access previous or next rows for comparisons.

### Syntax:

sql

Copy code

<function\_name> (<expression>) OVER (

[PARTITION BY <columns>] [ORDER BY <columns>] [ROWS/RANGE <frame\_specification>]

)

* PARTITION BY: Groups rows into partitions (optional).
* ORDER BY: Specifies the order of rows within the partition.
* **Frame Specification**: Defines the subset of rows for calculation (e.g., ROWS BETWEEN).

### Examples of Window Functions:

#### 1. ****Ranking Rows****

Using ROW\_NUMBER(), RANK(), and DENSE\_RANK():

sql

Copy code

SELECT name, salary,

ROW\_NUMBER() OVER (ORDER BY salary DESC) AS row\_num,

RANK() OVER (ORDER BY salary DESC) AS rank,

DENSE\_RANK() OVER (ORDER BY salary DESC) AS dense\_rank

FROM employees;

* ROW\_NUMBER(): Unique ranks.
* RANK(): Skips ranks for ties (e.g., 1, 2, 2, 4).
* DENSE\_RANK(): No skips in ranks (e.g., 1, 2, 2, 3).

#### 2. ****Running Totals****

Calculate cumulative salary per department:

sql

Copy code

SELECT department, employee, salary,

SUM(salary) OVER (PARTITION BY department ORDER BY salary) AS running\_total

FROM employees;

#### 3. ****Lagging and Leading****

Compare the salary of an employee with the previous and next employee:

sql

Copy code

SELECT name, salary,

LAG(salary) OVER (ORDER BY salary) AS previous\_salary,

LEAD(salary) OVER (ORDER BY salary) AS next\_salary

FROM employees;

#### 4. ****Percentile Rank****

Find the relative ranking of employees within departments:

sql

Copy code

SELECT name, salary, department,

PERCENT\_RANK() OVER (PARTITION BY department ORDER BY salary) AS percentile\_rank

FROM employees;

#### 5. ****Find the Second Highest Salary****

sql

Copy code

SELECT name, salary

FROM (

SELECT name, salary, ROW\_NUMBER() OVER (ORDER BY salary DESC) AS rank

FROM employees

) ranked

WHERE rank = 2;

### Benefits of Window Functions:

1. **Flexibility**: Combine aggregate results with individual rows.
2. **Performance**: More efficient than subqueries in many scenarios.
3. **Clarity**: Cleaner syntax for complex calculations.

Let’s use a hypothetical employees table to show the actual database structure, outputs for each query, and their explanations.

### Hypothetical Table: employees

| ****name**** | ****department**** | ****salary**** |
| --- | --- | --- |
| Alice | HR | 4000 |
| Bob | HR | 5000 |
| Carol | IT | 6000 |
| David | IT | 7000 |
| Eve | IT | 8000 |

### Query 1: Total Salary Per Department with Row-Level Details

sql

Copy code

SELECT name, department, salary,

SUM(salary) OVER (PARTITION BY department) AS total\_salary

FROM employees;

**Output:**

| ****name**** | ****department**** | ****salary**** | ****total\_salary**** |
| --- | --- | --- | --- |
| Alice | HR | 4000 | 9000 |
| Bob | HR | 5000 | 9000 |
| Carol | IT | 6000 | 21000 |
| David | IT | 7000 | 21000 |
| Eve | IT | 8000 | 21000 |

**Description:** Shows total salary per department while retaining each employee's row.

### Query 2: Rank Employees by Salary Within Each Department

sql

Copy code

SELECT department, name, salary,

RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS rank

FROM employees;

**Output:**

| ****department**** | ****name**** | ****salary**** | ****rank**** |
| --- | --- | --- | --- |
| HR | Bob | 5000 | 1 |
| HR | Alice | 4000 | 2 |
| IT | Eve | 8000 | 1 |
| IT | David | 7000 | 2 |
| IT | Carol | 6000 | 3 |

**Description:** Assigns ranks to employees in each department based on their salary.

### Query 3: Cumulative Salary by Department

sql

Copy code

SELECT department, name, salary,

SUM(salary) OVER (PARTITION BY department ORDER BY salary) AS running\_total

FROM employees;

**Output:**

| ****department**** | ****name**** | ****salary**** | ****running\_total**** |
| --- | --- | --- | --- |
| HR | Alice | 4000 | 4000 |
| HR | Bob | 5000 | 9000 |
| IT | Carol | 6000 | 6000 |
| IT | David | 7000 | 13000 |
| IT | Eve | 8000 | 21000 |

**Description:** Shows cumulative salary totals for each department in ascending order of salary.

### Query 4: Compare Salary with the Previous Row

sql

Copy code

SELECT name, salary,

LAG(salary) OVER (ORDER BY salary) AS previous\_salary

FROM employees;

**Output:**

| ****name**** | ****salary**** | ****previous\_salary**** |
| --- | --- | --- |
| Alice | 4000 | NULL |
| Bob | 5000 | 4000 |
| Carol | 6000 | 5000 |
| David | 7000 | 6000 |
| Eve | 8000 | 7000 |

**Description:** Shows each employee’s salary alongside the previous row’s salary (in ascending order).

### Query 5: Total and Average Salary in the Same Query

sql

Copy code

SELECT name, department, salary,

SUM(salary) OVER (PARTITION BY department) AS total\_salary,

AVG(salary) OVER (PARTITION BY department) AS avg\_salary

FROM employees;

**Output:**

| ****name**** | ****department**** | ****salary**** | ****total\_salary**** | ****avg\_salary**** |
| --- | --- | --- | --- | --- |
| Alice | HR | 4000 | 9000 | 4500 |
| Bob | HR | 5000 | 9000 | 4500 |
| Carol | IT | 6000 | 21000 | 7000 |
| David | IT | 7000 | 21000 | 7000 |
| Eve | IT | 8000 | 21000 | 7000 |

**Description:** Displays total and average salary per department alongside individual employee details.

1. when i use groupby i am going to get the values for each value instead of collapsing all rows then why i need window functions.

###### ChatGPT said:

You're correct that using GROUP BY can also perform aggregations for each group instead of collapsing all rows. However, **window functions** provide more flexibility and allow calculations across rows without grouping them into a single result. Here's why you might prefer window functions over GROUP BY in certain scenarios:

### Key Differences Between GROUP BY and Window Functions:

| ****Feature**** | GROUP BY | ****Window Functions**** |
| --- | --- | --- |
| **Row Output** | Collapses rows into groups, one row per group. | Retains individual rows while performing calculations. |
| **Use Case** | Aggregates data for groups. | Performs calculations across rows (e.g., ranking, cumulative totals, comparisons) without collapsing rows. |
| **Multiple Aggregates** | Not flexible; difficult for multiple aggregations. | Easily compute multiple aggregates in the same query. |
| **Scope** | Operates only on grouped rows. | Operates on a "window" of rows, which can include all or some rows. |
| **Complexity** | Simpler for group-level summaries. | Ideal for complex analytics like ranking, running totals, and comparisons. |

### Scenarios Where ****Window Functions**** Are Superior:

* 1. **Preserve Row Details While Aggregating**: With GROUP BY, you lose individual row details since rows are collapsed. Window functions let you retain all rows and show group-level aggregates alongside row-level details.

Example: Find the total salary for each department while keeping individual employee rows.

sql

Copy code

-- Using GROUP BY: You lose individual details.

SELECT department, SUM(salary) AS total\_salary

FROM employees

GROUP BY department;

-- Using Window Functions: Keep individual details with group total.

SELECT name, department, salary,

SUM(salary) OVER (PARTITION BY department) AS total\_salary

FROM employees;

* 1. **Rank/Order Within Groups**: Window functions like ROW\_NUMBER(), RANK(), or DENSE\_RANK() are designed for ranking rows within groups, which is not possible with GROUP BY.

Example: Rank employees by salary within each department.

sql

Copy code

SELECT department, name, salary,

RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS rank

FROM employees;

* 1. **Cumulative Totals**: Calculate running totals, averages, or percentages incrementally, which GROUP BY cannot do.

Example: Cumulative salary by department.

sql

Copy code

SELECT department, name, salary,

SUM(salary) OVER (PARTITION BY department ORDER BY salary) AS running\_total

FROM employees;

* 1. **Comparisons Across Rows**: Use LAG() or LEAD() to compare rows (e.g., previous or next row's value).

Example: Compare each employee's salary with their predecessor's:

sql

Copy code

SELECT name, salary,

LAG(salary) OVER (ORDER BY salary) AS previous\_salary

FROM employees;

* 1. **Multiple Aggregates Without Nesting Queries**: Window functions allow multiple aggregates without requiring nested queries.

Example: Find the total and average salary in the same query:

sql

Copy code

SELECT name, department, salary,

SUM(salary) OVER (PARTITION BY department) AS total\_salary,

AVG(salary) OVER (PARTITION BY department) AS avg\_salary

FROM employees;

### Why Use Window Functions?

* 1. **Row-level insight**: Analyze individual rows while computing aggregates or rankings.
  2. **Flexibility**: Perform complex calculations like rankings, cumulative sums, and comparisons across rows in a single query.
  3. **Efficiency**: Avoid creating additional subqueries for different metrics.

In summary, while GROUP BY is suitable for simple grouped aggregates, window functions shine when you need detailed insights, rankings, cumulative operations, or comparisons without collapsing rows.