WINDOW FUNCTIONS

The Complete Guide to SQL Window Functions: Ranking, Partitioning, and Beyond

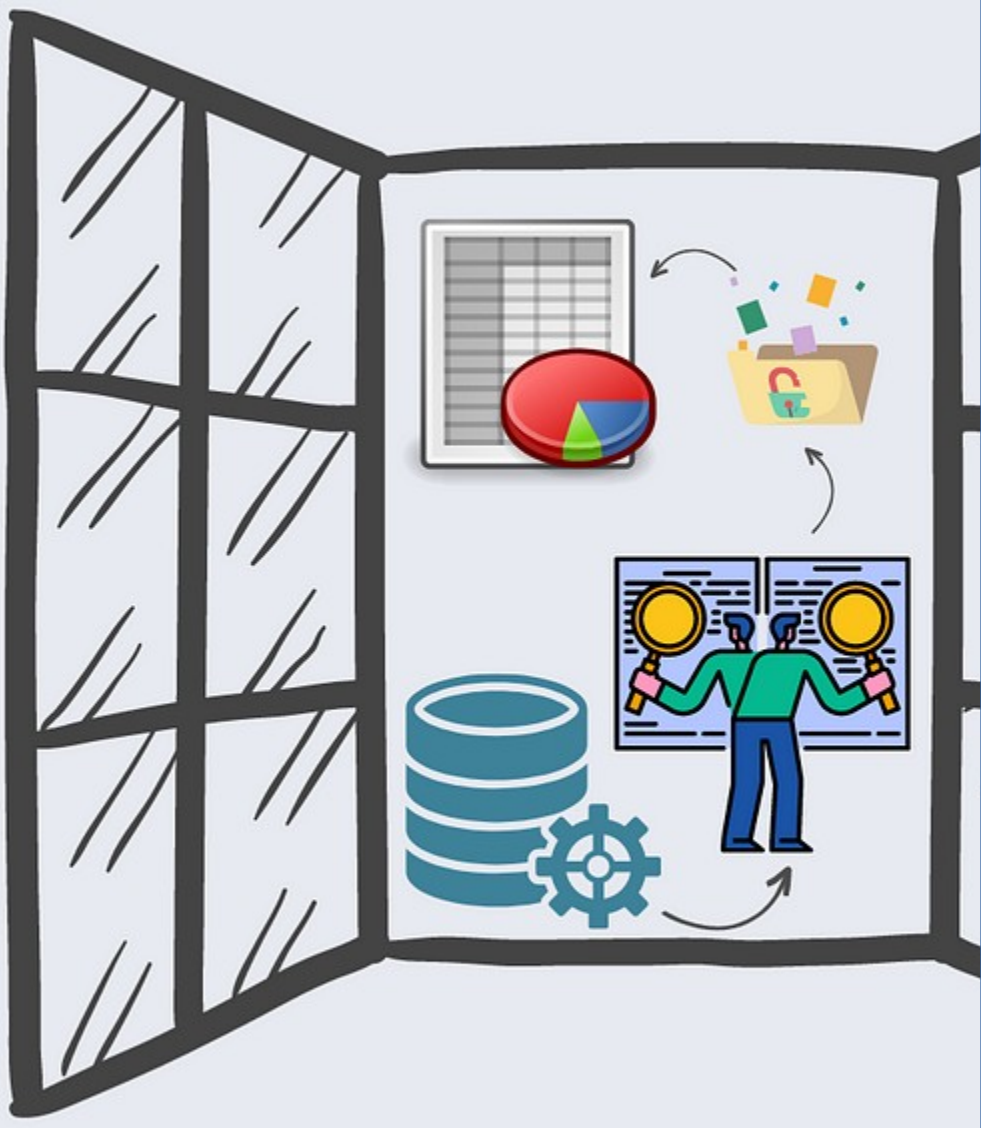
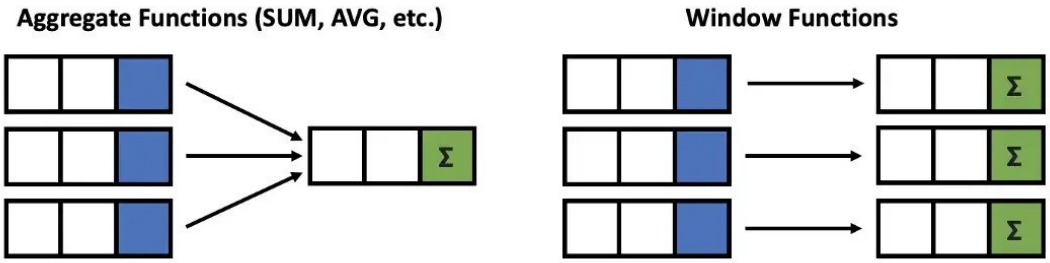


Table of Contents

INTRODUCTION:

A SQL Window Function is a function that performs a calculation across a set of rows related to the current row, while preserving the individual rows in the result.

Unlike aggregate functions, which group rows into a single output, window functions return a value for each row based on the defined window, using the `OVER()` clause to specify the partitioning and ordering of the rows within that window.



As from the name window in SQL is essentially a group of rows or observations within a table or result set. Depending on how you structure your query, you might define multiple windows within the same table, which you will explore further. A window is defined by using the OVER() clause in SQL.

Syntax:

```
      ----------------------------------------------------------------
Window_function(expression|column) OVER(
      [ PARTITION BY expr_list optional]
      [ ORDER BY order_list optional])
      ----------------------------------------------------------------
```

- **expression|column:** The target column for which the window function will compute a value.
- **OVER() clause:** Defines the window or set of rows on which the window function operates. Without this clause, the function operates on all rows in the result set.
- **PARTITION BY (Optional):** Divides the result set into partitions or sets, and the window function is applied to each partition separately. It works similarly to GROUP BY, but it doesn't reduce the number of rows.
    - SUM(salary) OVER (PARTITION BY department)
- **ORDER BY (Optional):** Specifies the order in which rows are processed within each partition. Useful for ranking or time-based calculations.

Why we need Windows Functions?

Suppose you are a teacher and want to analyze the performance of students in your class. You have a table called student_grades in your database that includes four columns student_id, student_name, subject, and grade.

Objective:

You want to:

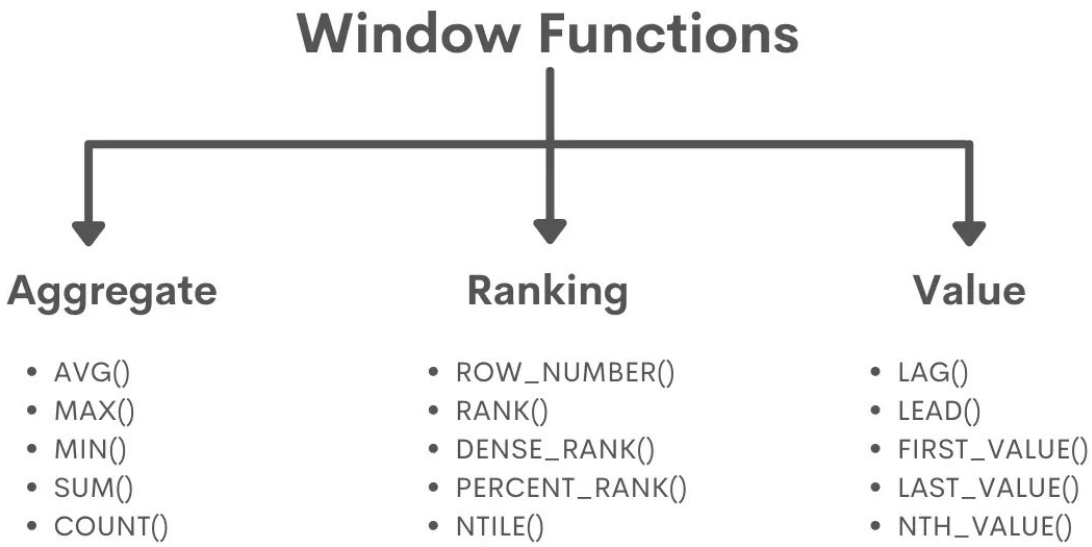
1. Assign a rank to each student based on their grades in each subject.
2. Calculate the average grade for each student across all subjects.
3. See the difference between each student's grade and the highest grade in the same subject.

student_id	student_name	subject	grade
1	Alice	Math	85.5
1	Alice	Science	92
1	Alice	English	88
2	Bob	Math	90
2	Bob	Science	78.5
2	Bob	English	85
3	Charlie	Math	75
3	Charlie	Science	80
3	Charlie	English	95
4	David	Math	85.5
4	David	Science	88
4	David	English	82

Without window functions, you would have to write multiple, complicated queries to rank students, calculate their average grades, and compare them with the top grades in each subject. This would be time-consuming and prone to errors. And it becomes simple and easy by using window functions.

TYPES OF WINDOW FUNCTIONS:

In SQL, window functions are classified into several types based on their purpose and functionality. Here's a brief overview of the types of window functions:



AGGREGATE WINDOW FUNCTION:

Aggregate window functions allow you to perform calculations (such as finding totals, averages, or counts) across a specific set of rows (a "window") while returning a result for each individual row in your dataset. Aggregate window functions combine the power of regular aggregate functions (like SUM, AVG, COUNT, etc.) with the flexibility of window functions.

AVG() WINDOW FUNCTION:

Suppose you want to know each student's grade and the average grade for each subject, but without losing any individual rows.

Table: student_grades

student_id	student_name	subject	grade
1	Alice	Math	85.5
2	Bob	Math	90.0
3	Charlie	Math	75.0
1	Alice	Science	92.0
2	Bob	Science	78.5
3	Charlie	Science	80.0

Query:

```
-------------------------------------------------------------
    SELECT
       student_id,
       student_name,
       subject,
       grade,
       AVG(grade) OVER (PARTITION BY subject) as avg_grade
    FROM
        student_grades;
-------------------------------------------------------------
```

Result:

student_id	student_name	subject	Grade	avg_grade
1	Alice	Math	85.5	83.50
2	Bob	Math	90.0	83.50
3	Charlie	Math	75.0	83.50
1	Alice	Science	92.0	83.50
2	Bob	Science	78.5	83.50
3	Charlie	Science	80.0	83.50

Explanation:

• AVG(grade) OVER (PARTITION BY subject): This calculates the average grade for each subject. The PARTITION BY subject clause groups the rows by subject and calculates the average within each group.

• Result: Every row keeps its individual data, but now each row also shows the average grade for that subject, making it easy to compare individual grades against the average.

MIN() and MAX() WINDOW FUNCTION:

• **MIN():** This function returns the smallest value in a set of rows. In a window function context, it finds the minimum value within a specified "window" or partition of rows, without reducing the total number of rows in the result set.

• **MAX():** This function returns the largest value in a set of rows. Similarly, in a window function context, it finds the maximum value within a specified window.

Example 01:

Suppose you have an employees table that includes columns for employee_id, department, and salary. You can use the MIN() and MAX() window functions to calculate the smallest and largest salaries within each department.

Table: employees

employee_id	employee_name	department	salary
1	Alice	IT	6000
2	Bob	IT	5000
3	Charlie	IT	7000
4	David	HR	4500
5	Eva	HR	5500
6	Frank	Finance	6200
7	Grace	Finance	7500
8	Henry	Finance	5900
9	Ivy	Marketing	4700
10	Jack	Marketing	5200

Query:

```
-----------------------------------------------------------------
SELECT employee_id as e_id,
department,
salary,
MIN(salary) OVER (PARTITION BY department) AS min_salary,
MAX(salary) OVER (PARTITION BY department) AS max_salary
FROM employees;
-----------------------------------------------------------------
```

Explanation:

- MIN(salary) OVER (PARTITION BY department): This calculates the minimum salary for each employee within their department.
- MAX(salary) OVER (PARTITION BY department): This calculates the maximum salary for each employee within their department.

Result:

e_id	employee_name	department	salary	min_salary	max_salary
1	Alice	IT	6000	5000	7000
2	Bob	IT	5000	5000	7000
3	Charlie	IT	7000	5000	7000
4	David	HR	4500	4500	5500
5	Eva	HR	5500	4500	5500
6	Frank	Finance	6200	5900	7500
7	Grace	Finance	7500	5900	7500
8	Henry	Finance	5900	5900	7500
9	Ivy	Marketing	4700	4700	5200
10	Jack	Marketing	5200	4700	5200

COUNT() WINDOW FUNCTION:

The **COUNT()** window function in SQL is used to count the number of rows within a specified "window" of rows, without collapsing the result set. It works similarly to the aggregate function COUNT(), but in this case, it counts rows within each partition while retaining each row in the output.

Example 01:

Consider the already discussed table employees table with columns employee_id, department, and salary.

Query:

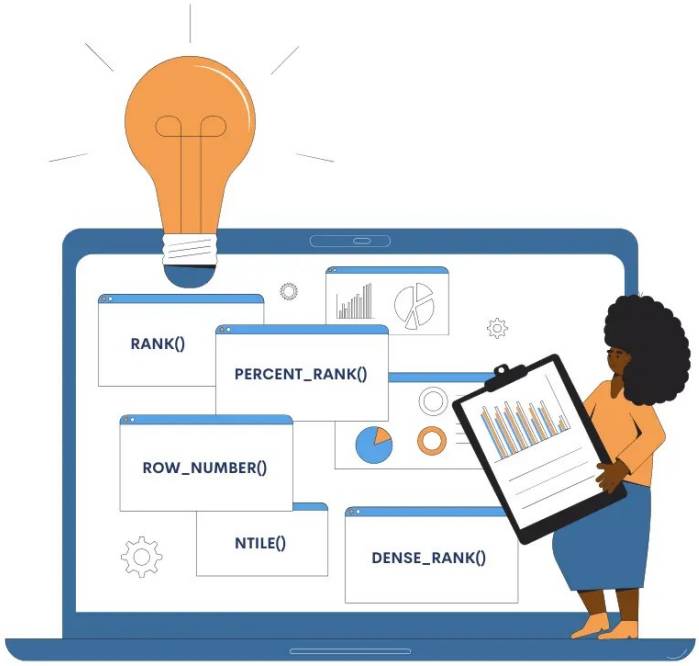
```
-----------------------------------------------------------------
SELECT

employee_id,

department,

COUNT(*) OVER (PARTITION BY department) AS total_employees
FROM employees;
-----------------------------------------------------------------
```

Result:

employee_id	department	total_employees
1	IT	3
2	IT	3
3	IT	3
4	HR	2
5	HR	2
6	Finance	3
7	Finance	3
8	Finance	3
9	Marketing	2
10	Marketing	2

RANKING WINDOW FUNCTION:

Ranking window functions in SQL assign a rank or position to each row within a specific set of rows (called a "window"). These functions are used to order rows and then assign a numerical rank based on that order. Unlike regular functions, they don’t group rows together; instead, they allow each row to retain its data while also showing its rank relative to others.



ROW_NUMBER() WINDOW FUNCTION:

The ROW_NUMBER() window function is used to assign a unique, sequential number to each row within a partition of a result set. It resets the numbering for each partition (if partitions are specified) and increments by 1 for every row within that partition or result set.

Example 01:

You have a table of employees with their department and salary, and you want to assign a unique row number to each employee within their department, ordered by salary.

EmployeeID	EmployeeName	Department	Salary
1	Alice	HR	4000
2	Bob	HR	3500
3	Charlie	IT	5000
4	David	IT	4500
5	Eve	HR	3000

Query:

```
-----------------------------------------------------------------
SELECT
    EmployeeID,
    EmployeeName,
    Department,
    Salary,
    ROW_NUMBER() OVER (PARTITION BY Department ORDER BY Salary DESC) AS
RowNum
FROM
    Employees;

-----------------------------------------------------------------
```

Result:

EmployeeID	EmployeeName	Department	Salary
1	Alice	HR	4000
2	Bob	HR	3500
5	Eve	HR	3000
3	Charlie	IT	5000
4	David	IT	4500

Explanation:

- The ROW_NUMBER() function assigns a unique row number to each employee within their respective department (PARTITION BY Department).
- The employees in each department are ordered by their salary in descending order (ORDER BY Salary DESC).
- The RowNum column shows the row number assigned to each employee within their department based on their salary.

Example 02:

You have a table of sales transactions, and you want to assign a unique row number to each transaction per customer, ordered by the date of the transaction (newest first).

TransactionID	CustomerID	TransactionDate	Amount
101	1	2024-09-25	500
102	1	2024-09-20	200
103	2	2024-09-21	700
104	2	2024-09-19	100
105	1	2024-09-22	300

Query:

```
-----------------------------------------------------------------
SELECT
    TransactionID,
    CustomerID,
    TransactionDate,
    Amount,
    ROW_NUMBER() OVER (PARTITION BY CustomerID ORDER BY TransactionDate
DESC) AS RowNum
FROM
    Sales;

-----------------------------------------------------------------
```

Result:

TransactionID	CustomerID	TransactionDate	Amount	RowNum
101	1	2024-09-25	500	1
105	1	2024-09-22	300	2
102	1	2024-09-20	200	3
103	2	2024-09-21	700	1
104	2	2024-09-19	100	2

Explanation:

- The ROW_NUMBER() function is used to assign a unique row number to each sales transaction, partitioned by the CustomerID.

- Within each partition, the transactions are ordered by the TransactionDate in descending order (ORDER BY TransactionDate DESC), so the most recent transaction gets the row number 1.
- The RowNum column provides the row number per customer, allowing you to track or rank their transactions by date.

RANK() WINDOW FUNCTION:

The RANK() window function assigns a rank to each row within a partition of a result set, based on the values in a specified column. Rows with equal values are given the same rank, but this causes gaps in the ranking for subsequent rows.

Example:

You have a table of students with their scores, and you want to rank them based on their scores in descending order. If two students have the same score, they should share the same rank, and the next rank should be skipped (like 1, 2, 2, 4).

EmployeeID	EmployeeName	Department	Salary
101	John	Sales	90000
102	Jane	Sales	75000
103	Jim	Sales	75000
104	Jack	IT	95000
105	Jill	IT	85000
106	Jeff	IT	85000
107	Jasmine	HR	80000
108	Joe	HR	75000
109	Jerry	HR	70000

Query:

```
-------------------------------------------------------------
SELECT
    EmployeeID,
    EmployeeName,
    Department,
    Salary,
    RANK() OVER (PARTITION BY Department ORDER BY Salary DESC) AS
SalaryRank
FROM
    Employees;

-------------------------------------------------------------
```

Result:

EmployeeID	EmployeeName	Department	Salary	SalaryRank
101	John	Sales	90000	1
102	Jane	Sales	75000	2
103	Jim	Sales	75000	2
104	Jack	IT	95000	1
105	Jill	IT	85000	2
106	Jeff	IT	85000	2
107	Jasmine	HR	80000	1
108	Joe	HR	75000	2
109	Jerry	HR	70000	3

Explanation:

- Partitioning by Department: We use PARTITION BY Department to rank employees within each department.
- Ordering by Salary: Employees are ranked within their departments in descending order of salary.
- Ties in Salary: Employees with the same salary (like Jane and Jim in Sales, or Jill and Jeff in IT) get the same rank.
- Rank Gaps: After ties, the next rank is skipped. For example, Jane and Jim both rank 2, so the next employee (if there were any more) would be ranked 4.

DENSE_RANK() WINDOW FUNCTION:

The DENSE_RANK() window function in SQL is used to rank rows within a result set or partition without skipping any ranks, even when there are ties. Unlike the RANK() function, which skips ranks for tied values, DENSE_RANK() assigns the same rank to tied rows but does not leave gaps in the ranking sequence.

Example :

You have a list of employees in different departments with their salaries, and you want to rank them within their department based on salary. Unlike RANK(), the DENSE_RANK() function will not skip ranks after ties.

EmployeeID	EmployeeName	Department	Salary
101	John	Sales	90000
102	Jane	Sales	75000
103	Jim	Sales	75000
104	Jack	IT	95000
105	Jill	IT	85000
106	Jeff	IT	85000
107	Jasmine	HR	80000
108	Joe	HR	75000
109	Jerry	HR	70000

Query:

```
-----------------------------------------------------------------
SELECT
    EmployeeID,
    EmployeeName,
    Department,
    Salary,
    DENSE_RANK() OVER (PARTITION BY Department ORDER BY Salary DESC) AS
SalaryRank
FROM
    Employees;

-----------------------------------------------------------------
```

Result:

EmployeeID	EmployeeName	Department	Salary	SalaryRank
101	John	Sales	90000	1
102	Jane	Sales	75000	2
103	Jim	Sales	75000	2
104	Jack	IT	95000	1
105	Jill	IT	85000	2
106	Jeff	IT	85000	2
107	Jasmine	HR	80000	1
108	Joe	HR	75000	2
109	Jerry	HR	70000	3

Explanation:

- Partitioning by Department: The DENSE_RANK() function partitions the data by Department, meaning that each department is ranked independently.
- Ordering by Salary: Employees are ranked within their department in descending order of their salary.
- Handling Ties: Unlike RANK(), DENSE_RANK() does not skip ranks when there are ties. For example, Jane and Jim in Sales both rank 2, and the next rank is 3 (instead of 4 as in RANK()).

DENSE_RANK() vs RANK() WINDOW FUNCTION:

RANK() window function assigns a rank to each row in the ordered result set. If two or more rows have the same value in the ORDER BY clause, they receive the same rank. However, RANK() skips ranks after ties, leaving gaps in the rank sequence.

Like RANK(), it assigns a rank to rows based on the specified ORDER BY clause. However, DENSE_RANK() does not skip ranks. When two or more rows tie, they get the same rank, but the next row gets the immediate subsequent rank.

PERCENT_RANK() WINDOW FUNCTION:

The PERCENT_RANK() window function calculates the relative rank of a row as a percentage between 0 and 1 within its result set or partition. It is used to find the percentile ranking of a row compared to other rows.

How PERCENT_RANK() Works:

- **Formula**:
  - PERCENT_RANK=Rank of Current Row−1 / Total Rows−1
  - The formula shows that the function calculates the rank relative to the size of the result set, normalized between 0 and 1.
  - If the current row has the lowest value, the PERCENT_RANK() will return 0.
  - If the current row has the highest value, it will return 1 (or close to 1 depending on the number of rows).
- **Partitioning**: If the PARTITION BY clause is used, the function calculates the percentage rank for each partition individually.

Example:

You have a dataset of students and their test scores, and you want to calculate the percentile rank of each student's score. The PERCENT_RANK() function calculates the relative rank of a row within a partition of data, expressing the rank as a percentage between 0 and 1. It is based on the rank position in relation to the total number of rows.

StudentID	StudentName	Score
1	Alice	85
2	Bob	95
3	Charlie	85
4	David	70
5	Eve	90
6	Frank	75

Query:

```
-----------------------------------------------------------------
SELECT
    StudentID,
    StudentName,
    Score,
    PERCENT_RANK() OVER (ORDER BY Score DESC) AS PercentRank
FROM
    Students;

-----------------------------------------------------------------
```

Result:

StudentID	StudentName	Score	PercentRank
2	Bob	95	0
5	Eve	90	0.2
1	Alice	85	0.4
3	Charlie	85	0.4
6	Frank	75	0.8
4	David	70	1

In summary, **PERCENT_RANK()** is a useful function for determining the relative position of a row in a dataset, expressed as a percentage. It is particularly helpful when analyzing the distribution of values across a dataset.

NTILE() WINDOW FUNCTION:

The NTILE() window function divides rows into a specified number of roughly equal-sized groups or buckets, assigning each row a "bucket number." This is useful for distributing data evenly into groups, like quartiles or percentiles.

Example:

Let's use a dataset of students and their grades to demonstrate the use of the NTILE() function. In this case, we will divide the students into 3 groups (tertiles) based on their grades.

student_id	name	grade
1	Alice	90
2	Bob	80
3	Charlie	85
4	David	95
5	Eva	70
6	Frank	75
7	Grace	60
8	Helen	65

Query:

```
-----------------------------------------------------------------
SELECT
student_id,
name,
grade,
NTILE(3) OVER (ORDER BY grade DESC) AS grade_group
FROM students;
-----------------------------------------------------------------
```

Result:

student_id	name	grade	grade_group
4	David	95	1
1	Alice	90	1
3	Charlie	85	1
2	Bob	80	2
6	Frank	75	2
5	Eva	70	3
8	Helen	65	3
7	Grace	60	3

Explanation:

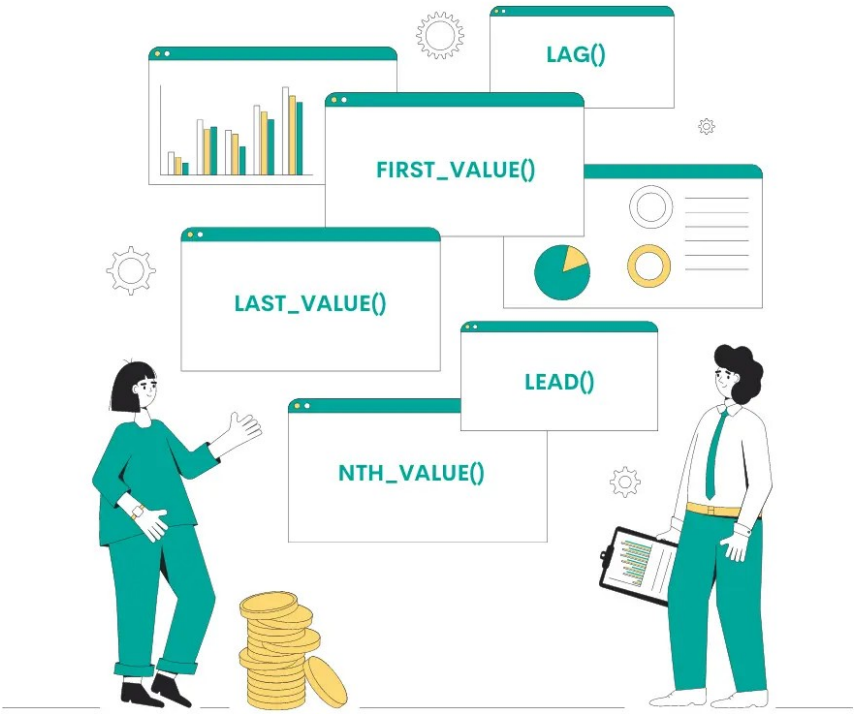
- The NTILE(3) function divides the 8 students into **3 equal groups** based on their grades.
- The **first group** (grade_group = 1) contains the students with the highest grades: David, Alice, and Charlie.
- The **second group** (grade_group = 2) contains students with mid-range grades: Bob and Frank.
- The **third group** (grade_group = 3) contains the students with the lowest grades: Eva, Helen, and Grace.

Use Case:

This type of analysis is often used in education to divide students into **percentiles or groups** based on performance, helping teachers understand the distribution of grades and students' performance relative to each other.

VALUE() WINDOW FUNCTION:

The **VALUE( )** window functions in SQL refer to specific window functions that retrieve values from certain positions within a window frame of rows. These include:



LEAD() WINDOW FUNCTION:

The **LEAD()** function allows you to access data from the following (next) row relative to the current row in the result set. It helps in comparing a row with future rows, making it useful for operations like calculating differences over time.

Syntax:

```
-----------------------------------------------------------------
LEAD(column_name, offset, default_value)
OVER ( [ PARTITION BY expr_list optional ]
ORDER BY order_list )

-----------------------------------------------------------------
```

- **`column_name`**: The column from which to fetch the value.
- **`offset`**: Number of rows ahead from the current row (default is 1).
- **`default_value`**: Value to return if the offset goes beyond the dataset (optional).

Example:

Suppose we have a table of sales data.

month	sales
January	1000
February	1200
March	1500
April	1700

Query:

```
SELECT month, sales,
LEAD(sales) OVER (ORDER BY month) AS next_month_sales
FROM sales_data;
```

Result:

month	sales	next_month_sales
January	1000	1200
February	1200	1500
March	1500	1700
April	1700	NULL

The LEAD() function retrieves the sales value for the next month. In April, there is no next month, so it returns NULL.

LAG() WINDOW FUNCTION:

The **LAG()** function is the opposite of LEAD(). It allows you to access data from a preceding (previous) row relative to the current row. It's useful for comparing the current row with past values.

Syntax:

```
LAG(column_name, offset, default_value)
OVER ( [ PARTITION BY expr_list optional ]
ORDER BY order_list )
```

- **`column_name`**: The column from which to fetch the value.
- **`offset`**: Number of rows behind from the current row (default is 1).
- **`default_value`**: Value to return if the offset goes beyond the dataset (optional).

Example:

If we calcuate the previous month sales using the LAG() window function for the same previous example:

Query:

```
SELECT month, sales,
LAG(sales) OVER (ORDER BY month) AS previous_month_sales
FROM sales_data;
```

Result:

month	sales	previous_month_sales
January	1000	NULL
February	1200	1000
March	1500	1200
April	1700	1500

The LAG() function fetches the sales value from the previous month. In January, there is no previous month, so it returns NULL

Practical Use Cases:

- **LEAD()**: Useful for comparing future values, such as stock prices, sales, or trends.
- **LAG()**: Commonly used to calculate month-over-month differences, year-over-year growth, or trend comparisons with past data.

Key Differences:

- **LEAD()** fetches data from the next row, while **LAG()** fetches from the previous row.
- Both are useful for **time series analysis**, trend comparisons, or sequential data exploration.

FIRST_VALUE() WINDOW FUNCTION:

Both **FIRST_VALUE()** and **LAST_VALUE()** window functions are used to retrieve the **first** or **last** value in a window of ordered rows. These functions are useful for accessing the first or last data point within a partition or an entire dataset.

The **FIRST_VALUE()** function returns the first value in the window frame based on the specified ordering.

Syntax:

```
-----------------------------------------------------------------
FIRST_VALUE(column_name) OVER ( [PARTITION BY expr_list optional]
ORDER BY order_list [ROWS or RANGE frame_clause optional] )

-----------------------------------------------------------------
```

- **column_name**: The column from which to return the first value.
- **PARTITION BY**: (Optional) Divides the result set into partitions. FIRST_VALUE() is applied separately to each partition.
- **ORDER BY**: Defines the order of rows in the window.
- Frame Clause: (Optional) Controls how many rows are considered in the window frame.

Example:

Let's take a dataset of employees and their salaries:

employee_id	name	salary
1	Alice	6000
2	Bob	5000
3	Charlie	7000
4	David	4500
5	Eva	5500

Query:

```
-----------------------------------------------------------------
SELECT employee_id, name, salary,
FIRST_VALUE(salary) OVER (ORDER BY salary DESC) AS highest_salary
FROM employees;
-----------------------------------------------------------------
```

Result:

employee_id	name	salary	highest_salary
3	Charlie	7000	7000
1	Alice	6000	7000
5	Eva	5500	7000

2	Bob	5000	7000
4	David	4500	7000

The FIRST_VALUE() function fetches the first value based on the order of salaries. Since the ORDER BY salary DESC is used, it fetches the highest salary, which is 7000.

LAST_VALUE() WINDOW FUNCTION:

The **LAST_VALUE()** function returns the last value in the window frame based on the specified ordering.

Syntax:

```
-----------------------------------------------------------------
LAST_VALUE(column_name) OVER ( [PARTITION BY expr_list optional]
ORDER BY order_list [ROWS or RANGE frame_clause optional] )

-----------------------------------------------------------------
```

- **column_name**: The column from which to return the last value.
- **PARTITION BY**: (Optional) Divides the result set into partitions. LAST_VALUE() is applied separately to each partition.
- **ORDER BY**: Defines the order of rows in the window.
- **Frame Clause**: (Optional) Controls how many rows are considered in the window frame.

Using the same employee dataset, let's retrieve the last salary in the ordered list:

Query:

```
-----------------------------------------------------------------
SELECT employee_id, name, salary,
LAST_VALUE(salary) OVER (ORDER BY salary DESC ROWS BETWEEN UNBOUNDED
PRECEDING AND UNBOUNDED FOLLOWING) AS last_salary
FROM employees;
-----------------------------------------------------------------
```

Result:

employee_id	name	salary	last_salary
3	Charlie	7000	4500
1	Alice	6000	4500
5	Eva	5500	4500
2	Bob	5000	4500
4	David	4500	4500

- The LAST_VALUE() function fetches the **last salary** (4500) based on the descending order of salaries.

- The frame clause ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING is used to ensure that the function considers all rows in the partition.

Key Differences Between FIRST_VALUE() and LAST_VALUE():

1. **FIRST_VALUE()** returns the first value in the window based on the specified order.
2. **LAST_VALUE()** returns the last value in the window based on the specified order.
3. **Frame Clauses**: If you don't specify the frame clause properly with LAST_VALUE(), it may not behave as expected. This is because SQL window functions operate on a window frame, and by default, the window only considers rows up to the current row unless specified otherwise.

Practical Use Cases:

- **FIRST_VALUE()**: Useful for retrieving the earliest event in a time series, such as the first sale, the first payment, etc.
- **LAST_VALUE()**: Useful for retrieving the most recent data, like the last transaction or the most recent sale.

Nth_VALUE() WINDOW FUNCTION:

The **NTH_VALUE()** window function allows you to retrieve the value from a specific row, based on its position, within a window or partition. It is useful when you need to access not just the first or last value, but any specific row in the dataset.

Syntax:

```
-----------------------------------------------------------------
NTH_VALUE(column_name, N) OVER ( [PARTITION BY expr_list optional]
ORDER BY order_list [ROWS or RANGE frame_clause optional] )
-----------------------------------------------------------------
```

- **column_name**: The column from which to retrieve the value.
- **N**: The position of the value you want to retrieve. For example, N=2 fetches the 2nd value.
- **PARTITION BY**: (Optional) Divides the result set into partitions. NTH_VALUE() is applied separately within each partition.
- **ORDER BY**: Specifies the order in which rows are processed within the window.
- **Frame Clause**: (Optional) Defines the subset of rows for the window function to consider.

How it works?

- **NTH_VALUE(column_name, N)** returns the value of the **N-th row** within the ordered window.

- Unlike **FIRST_VALUE()** and **LAST_VALUE()**, which access the first and last rows respectively, NTH_VALUE() can retrieve any specific row based on its position in the window.
- The **frame clause** is critical in NTH_VALUE() because, by default, only rows up to the current row are considered. You may need to specify a frame like UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING to include all rows.

Example:

Let's consider a dataset of salespeople and their sales amounts:

Table: sales_data

employee_id	name	sales
1	Alice	9000
2	Bob	8000
3	Charlie	7000
4	David	8500
5	Eva	9500

Query:

```
SELECT employee_id, name, sales,
NTH_VALUE(sales, 3) OVER (ORDER BY sales DESC
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS
third_highest_sale
FROM sales_data;
```

Result:

employee_id	name	sales	third_highest_sale
5	Eva	9500	8500
1	Alice	9000	8500
4	David	8500	8500
2	Bob	8000	8500
3	Charlie	7000	8500

Explanation:

- The NTH_VALUE(sales, 3) function returns the value of the third-highest sales amount (8500).

- We used the **frame clause** ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING to ensure that the function considers all rows when determining the third-highest sale.

Key Points:

1. **N**: Specifies which row you want to retrieve within the window (e.g., 2nd, 3rd, etc.).
2. **Order Matters**: NTH_VALUE() relies heavily on the **ORDER BY** clause to determine the row position.
3. **Frame Clause**: It's important to use the frame clause carefully, as the default frame only considers rows from the start up to the current row.
4. **Use Cases**: Retrieving specific rankings, such as the 2nd or 3rd highest/lowest value, from a list of data.

Practical Use Cases:

- **Financial Reports**: To access the 2nd, 3rd, or 4th highest sales/revenue figures in a company's dataset.
- **Rankings**: For student scores, sports tournaments, or performance evaluations where intermediate positions matter.
- **Comparative Analysis**: Comparing the top N values in a dataset for analytical purposes.

Practice Problems:

Q1: You have two tables: 'response_times' with columns (request_id, response_time_ms, device_type_id) and 'device_types' with columns (device_type_id, device_name, manufacturer). Write a query to calculate the 95th percentile of response times for each device manufacturer.

Q2: Given a table 'daily_visits' with columns (visit_date, visit_count), write a query to calculate the 7-day moving average of daily visits for each date.

Q3: Given a table 'stock_prices' with columns (date, stock_symbol, closing_price). What's the cumulative change in stock price compared to the starting price of the year?

Q4: You have two tables: 'products' with columns (product_id, product_name, category_id, price) and 'categories' with columns (category_id, category_name). What is the price difference between each product and the next most expensive product in that category?

Q5: Given a table 'customer_spending' with columns (customer_id, total_spend), how would you divide customers into 10 deciles based on their total spending?

Q6: Using a table 'daily_active_users' with columns (activity_date, user_count), write a query to calculate the day-over-day change in user count and the growth rate.

Q7: Given a table 'sales' with columns (sale_id, sale_date, amount), how would you calculate the total sales amount for each day of the current month, along with a running total of month-to-date sales?

Q8: You have two tables 'employee_sales' with columns (employee_id, department_id, sales_amount) and ‘employees’ with columns (employee_id, employee_name), write a query to identify the top 5 employees by sales amount in each department.

Q9: Using a table 'employee_positions' with columns (employee_id, position, start_date, end_date), write a query to find employees who have been promoted (i.e., changed to a different position) within 6 months of their initial hire.

Q10: You have two tables: 'customer_transactions' with columns (customer_id, transaction_date, transaction_amount), and 'customer_info' with columns (customer_id, customer_name, signup_date). Write a query to calculate the moving average of transaction amounts for each customer over their last 3 transactions, only for customers who signed up for more than a year.