What is a Window Function?

A **window function** is a type of SQL function that performs calculations across a set of rows related to the current row within a specified window or "frame" of data. Unlike aggregate functions, which return a single result for a set of rows, window functions preserve each row's original data and add a calculated column based on that data.

Characteristics of Window Functions

- **Operates on a "window" of rows**: It performs calculations on a subset of rows defined by the `OVER()` clause, which can include the entire dataset or be limited to specific partitions.
- **Doesn't require `GROUP BY`**: Unlike aggregate functions that reduce rows into a single result per group, window functions allow calculations across individual rows within a frame, making them ideal for row-level insights.
- **Preserves original rows**: Rows remain intact, and a new calculated column is added without reducing the row count.

Structure of a Window Function

The general syntax of a window function is as follows:

```
window_function() OVER ([PARTITION BY column_name] [ORDER BY
column_name] [window_frame])
```

Components:

1. **Window Function**: Any of several functions, including `RANK()`, `ROW_NUMBER()`, `DENSE_RANK()`, `SUM()`, `AVG()`, `MIN()`, `MAX()`, etc.

2. **OVER Clause**:

   - **PARTITION BY**: Divides the data into partitions or groups where the window function is applied. Each partition can have its ranking or calculations.
   - **ORDER BY**: Determines the order within each partition. For ranking functions, this order directly affects the rank assigned to each row.
   - **Window Frame** (optional): Defines a subset of rows within each partition to be used for calculations. For example, you can use `ROWS BETWEEN 1 PRECEDING AND CURRENT ROW` to include only certain rows within the partition.

Types of Window Functions

1. **Aggregate Window Functions**: `SUM()`, `AVG()`, `MIN()`, `MAX()`, `COUNT()` calculate totals or statistics within a partition.

2. **Ranking Functions**: `ROW_NUMBER()`, `RANK()`, `DENSE_RANK()`, `NTILE()` assign ranks or positions based on a column's ordered values.

3. **Value Functions**: `LEAD()`, `LAG()`, `FIRST_VALUE()`, `LAST_VALUE()` allow you to access previous or following rows' values in the dataset.

Example of a Window Function with `RANK()`

Using the `student_table`, we can calculate the rank of students based on their `stu_marks` within each subject (`stu_sub`) group:

```
SELECT *,
    RANK() OVER (PARTITION BY stu_sub ORDER BY stu_marks DESC) AS
Subject_Rank
FROM student_table;
```

This will rank students within each subject, starting with the highest `stu_marks`.

stu_name	stu_age	stu_add	stu_sub	stu_marks	Subject_Rank
John	22	Columbus	AI	89	1
Anthony	20	Austin	AI	82	2
Lily	23	Houston	AI	3	
Nicole	22	Dallas	Network	86	1
Lucy	24	Columbus	Network	2	

Complete Data Table (student_table)

stu_name	stu_age	stu_add	stu_sub	stu_marks
John	22	Columbus	AI	89
Nicole	22	Dallas	Network	86
Anthony	20	Austin	AI	82
James	20	Chicago	DAA	82
Tom	28	Naperville	ML	80
Lily	23	Houston	AI	75
Bob	25	Chicago	DAA	70
Emma	21	Dallas	ML	65
Lucy	24	Columbus	Network	60
Mark	22	Austin	Network	55

Applying Ranking Functions

We'll apply ranking functions to the `student_table` dataset, ordering by `stu_marks` in descending order to rank students based on their scores. Here's how each ranking function works and the output it generates.

Key Points:

- **Used with `OVER()` clause**: Ranking functions require the `OVER()` clause, which defines the partition and order for ranking.
- **Ranks assigned based on `ORDER BY` clause**: The function ranks rows in sequential order, starting at 1 within each partition.
- **Ranks reset with new partitions**: Each partition starts ranking again from 1.

---

1. `ROW_NUMBER()` Example

The `ROW_NUMBER()` function provides a unique, sequential rank to each row within a partition, regardless of duplicate values.

Syntax:

```
SELECT *, ROW_NUMBER() OVER(PARTITION BY column_name ORDER BY
column_name [ASC/DESC]) AS 'Position'
FROM table_name;
```

Example:

```
SELECT *,
    ROW_NUMBER() OVER (ORDER BY stu_marks DESC) AS Position
FROM student_table;
```

stu_name	stu_age	stu_add	stu_sub	stu_marks	Position
John	22	Columbus	AI	89	1
Nicole	22	Dallas	Network	86	2
Anthony	20	Austin	AI	82	3
James	20	Chicago	DAA	82	4
Tom	28	Naperville	ML	80	5
Lily	23	Houston	AI	75	6
Bob	25	Chicago	DAA	70	7
Emma	21	Dallas	ML	65	8
Lucy	24	Columbus	Network	60	9
Mark	22	Austin	Network	55	10

Explanation: `ROW_NUMBER()` assigns a unique, sequential number to each row ordered by `stu_marks` in descending order, without considering ties.

---

2. `RANK()` Example

The `RANK()` function assigns ranks with gaps. When rows have identical values, they receive the same rank, and the next rank skips by the count of tied rows.

Syntax:

```
SELECT *, RANK() OVER(PARTITION BY column_name ORDER BY
column_name [ASC/DESC]) AS 'Position'
FROM table_name;
```

Example:

```
SELECT *,
    RANK() OVER (ORDER BY stu_marks DESC) AS Position
FROM student_table;
```

stu_name	stu_age	stu_add	stu_sub	stu_marks	Position
John	22	Columbus	AI	89	1
Nicole	22	Dallas	Network	86	2
Anthony	20	Austin	AI	82	3
James	20	Chicago	DAA	82	3
Tom	28	Naperville	ML	80	5
Lily	23	Houston	AI	75	6
Bob	25	Chicago	DAA	70	7
Emma	21	Dallas	ML	65	8
Lucy	24	Columbus	Network	60	9
Mark	22	Austin	Network	55	10

Explanation: `RANK()` ranks students by `stu_marks`, assigning equal ranks to tied scores and leaving gaps (here, it skips 4).

If two students have identical marks, they receive the same rank, and the next rank will skip ahead by the count of duplicates.

---

3. `DENSE_RANK()` Example

The `DENSE_RANK()` function also assigns the same rank to duplicate values, but without gaps. The next rank in sequence will follow the last rank used, regardless of ties.

Syntax:

```
SELECT *, DENSE_RANK() OVER(PARTITION BY column_name ORDER BY
column_name [ASC/DESC]) AS 'Position'
FROM table_name;
```

Example:

```
SELECT *,
    DENSE_RANK() OVER (ORDER BY stu_marks DESC) AS Position
FROM student_table;
```

stu_name	stu_age	stu_add	stu_sub	stu_marks	Position
John	22	Columbus	AI	89	1
Nicole	22	Dallas	Network	86	2
Anthony	20	Austin	AI	82	3
James	20	Chicago	DAA	82	3
Tom	28	Naperville	ML	80	4
Lily	23	Houston	AI	75	5
Bob	25	Chicago	DAA	70	6
Emma	21	Dallas	ML	65	7

stu_name	stu_age	stu_add	stu_sub	stu_marks	Position
Lucy	24	Columbus	Network	60	8
Mark	22	Austin	Network	9	9

Explanation: `DENSE_RANK()` assigns equal ranks to ties without leaving gaps, maintaining consecutive ranks.

What Are Ranking Functions?

Ranking functions are a type of window function used to assign a unique ranking position to each row based on a specified column, often ordered by numeric or date values. Ranking functions are useful in analytics and reporting to establish ordered or hierarchical views of data.

Types of Ranking Functions:

1. **`ROW_NUMBER()`** : Assigns a unique number to each row, even if there are duplicate values.
2. **`RANK()`** : Assigns the same rank to duplicate values, leaving gaps in the ranking.
3. **`DENSE_RANK()`** : Similar to `RANK()`, but without gaps between ranks.
4. **`NTILE(n)`** : Divides rows into `n` equally sized groups and assigns each row a group number.

Conditions of Ranking Functions

To use ranking functions effectively, consider these conditions:

1. **Order**: Ranking requires an `ORDER BY` clause to determine how rows are ranked. Rankings depend on the specified column's values in ascending or descending order.

2. **Partitioning**: Ranking functions can be partitioned to reset the ranking within groups. The `PARTITION BY` clause groups rows by a specified column(s), and ranking is applied within each partition.

3. **Handling Ties**:

   - **ROW_NUMBER()**: Ignores ties, providing unique ranks.
   - **RANK()**: Ranks duplicate values equally and leaves gaps.
   - **DENSE_RANK()**: Ranks duplicate values equally without gaps.

Summary of Output

- **ROW_NUMBER()** provides a unique rank for each row without considering duplicates.
- **RANK()** accounts for ties but skips ranks for duplicate scores.
- **DENSE_RANK()** also considers ties but does not skip ranks, maintaining consecutive numbering.
- **NTILE()**: Divides data into specified groups.

Window Function Structure

```
window_function() OVER ([PARTITION BY column_name] [ORDER BY
column_name] [window_frame])
```

- **PARTITION BY**: Divides rows into groups (partitions) for function application.
- **ORDER BY**: Determines the order of rows within each partition.
- **Window Frame**: Defines the range of rows for function calculations within each partition (optional).

why we need window functions and where they can be applied, along with real-life examples.

Why We Need Window Functions

1. **Perform Calculations on a Set of Rows**: Window functions allow calculations across rows without reducing the number of rows in the result set. This is useful for comparative analysis and trend observation.

2. **Maintain Row-Level Detail**: Unlike aggregate functions that group rows, window functions preserve the original detail of each row while allowing for advanced analytics.

3. **Flexible Analytics**: They can be used for various analyses, including running totals, moving averages, ranking, and more, providing flexibility in data analysis.

4. **Simplified Query Structure**: Using window functions can often lead to simpler and more readable SQL queries than subqueries or self-joins.

Where to Use Window Functions

1. **Running Totals**: Calculate a cumulative sum across rows.

   - **Example**: In a sales database, you want to calculate a running total of sales for each month.

     ```
     SELECT
         month,
         sales,
         SUM(sales) OVER (ORDER BY month) AS running_total
     FROM sales_data;
     ```

2. **Moving Averages**: Calculate averages over a specific range of rows.

   - **Example**: For a stock price database, calculate a 3-month moving average.

     ```
     SELECT
         date,
         stock_price,
         AVG(stock_price) OVER (ORDER BY date ROWS BETWEEN 2
     PRECEDING AND CURRENT ROW) AS moving_average
     FROM stock_prices;
     ```

3. **Ranking**: Assign ranks to rows based on specific criteria.

    - **Example**: In a sports database, rank players based on their scores.

    ```sql
    SELECT
        player_name,
        score,
        RANK() OVER (ORDER BY score DESC) AS player_rank
    FROM player_scores;
    ```

4. **Percentiles**: Calculate percentile ranks.

    - **Example**: Determine the percentile of student scores in an exam.

    ```sql
    SELECT
        student_name,
        score,
        PERCENT_RANK() OVER (ORDER BY score) AS percentile
    FROM student_scores;
    ```

5. **Comparative Analysis**: Compare a row to others in its partition.

    - **Example**: Compare each employee's salary to the average salary in their department.

    ```sql
    SELECT
        employee_name,
        department,
        salary,
        AVG(salary) OVER (PARTITION BY department) AS
    department_average
    FROM employees;
    ```

6. **Data Segmentation**: Segment data into equal parts.

    - **Example**: Divide a list of customers into quartiles based on their purchase amounts.

    ```sql
    SELECT
        customer_name,
        purchase_amount,
        NTILE(4) OVER (ORDER BY purchase_amount) AS quartile
    FROM customer_purchases;
    ```

Real-Life Use Cases

1. **Financial Reporting**: Banks and financial institutions use window functions to calculate monthly or yearly totals and averages, which can help in budget forecasting and performance analysis.

2. **Sales Performance Tracking**: Retail companies often analyze sales trends over time, applying running totals or moving averages to evaluate performance against targets.

3. **Healthcare Analytics**: In healthcare, patient records can be analyzed to determine treatment effectiveness over time, using moving averages or cumulative counts to track outcomes.

4. **HR Analytics**: Organizations can assess employee performance and turnover rates by ranking employees based on performance metrics and comparing them against departmental averages.

5. **Web Analytics**: Websites use window functions to analyze user engagement metrics, such as average session duration or bounce rates over specified periods, providing insights into user behavior.

```
In [ ]:
```