# Window Functions and Advanced Analytics in SQL

## Objective:

This manual covers advanced SQL operations, focusing on CTEs and window functions. It provides detailed explanations and examples to help you understand and apply these concepts in real-world scenarios.

- Understand window functions and explain their use cases.
- Use various window functions such as `ROW_NUMBER`, `RANK`, and `DENSE_RANK` to perform complex calculations over a set of table rows.
- Demonstrate practical examples of window functions to rank data and analyze partitions.

## Working with Window Functions

### Introduction to Window Functions

Window functions perform calculations across a set of table rows related to the current row. Unlike aggregate functions, they do not collapse the rows into a single result, allowing detailed analytical operations over query results.

### Syntax of Window Functions

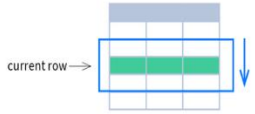
```
function_name([arguments]) OVER (
    [PARTITION BY partition_expression]
    [ORDER BY sort_expression]
    [frame_clause]
)
```

### Components:

- **function_name**: The name of the window function (e.g., `RANK`, `DENSE_RANK`, `ROW_NUMBER`).
- **arguments**: Function-specific arguments.
- **OVER**: Specifies the window.
- **PARTITION BY**: Divides the result set into partitions.
- **ORDER BY**: Orders the rows within each partition.
- **frame_clause**: Defines a subset of rows within the partition.

### WINDOW FUNCTIONS

compute their result based on a sliding window frame, a set of rows that are somehow related to the current row.




### AGGREGATE FUNCTIONS VS. WINDOW FUNCTIONS

unlike aggregate functions, window functions do not collapse rows.


### SYNTAX

```
SELECT city, month,
   sum(sold) OVER (
      PARTITION BY city
      ORDER BY month
      RANGE UNBOUNDED PRECEDING) total
FROM sales;
```

```
SELECT <column_1>, <column_2>,
   <window_function>() OVER (
      PARTITION BY <...>
      ORDER BY <...>
      <window_frame>) <window_column_alias>
FROM <table_name>;
```

### Named Window Definition

```
SELECT country, city,
     rank() OVER country_sold_avg
FROM sales
WHERE month BETWEEN 1 AND 6
GROUP BY country, city
HAVING sum(sold) > 10000
WINDOW country_sold_avg AS (
   PARTITION BY country
   ORDER BY avg(sold) DESC)
ORDER BY country, city;
```

```
SELECT <column_1>, <column_2>,
   <window_function>() OVER <window_name>
FROM <table_name>
WHERE <...>
GROUP BY <...>
HAVING <...>
WINDOW <window_name> AS (
   PARTITION BY <...>
   ORDER BY <...>
   <window_frame>)
ORDER BY <...>;
```

PARTITION BY, ORDER BY, and window frame definition are all optional.

### LOGICAL ORDER OF OPERATIONS IN SQL

1. FROM, JOIN
2. WHERE
3. GROUP BY
4. aggregate functions
5. HAVING
6. **window functions**
7. SELECT
8. DISTINCT
9. UNION/INTERSECT/EXCEPT
10. ORDER BY
11. OFFSET
12. LIMIT/FETCH/TOP

You can use window functions in SELECT and ORDER BY. However, you can't put window functions anywhere in the FROM, WHERE, GROUP BY, or HAVING clauses.

### PARTITION BY

divides rows into multiple groups, called **partitions**, to which the window function is applied.

| month | city | sold |
|---|---|---|
| 1 | Rome | 200 |
| 2 | Paris | 500 |
| 1 | London | 100 |
| 1 | Paris | 300 |
| 2 | Rome | 300 |
| 2 | London | 400 |
| 3 | Rome | 400 |

PARTITION BY city

| month | city | sold | sum |
|---|---|---|---|
| 1 | Paris | 300 | 800 |
| 2 | Paris | 500 | 800 |
| 1 | Rome | 200 | 900 |
| 2 | Rome | 300 | 900 |
| 3 | Rome | 400 | 900 |
| 1 | London | 100 | 500 |
| 2 | London | 400 | 500 |

**Default Partition:** with no PARTITION BY clause, the entire result set is the partition.

### ORDER BY

specifies the order of rows in each partition to which the window function is applied.

| sold | city | month |
|---|---|---|
| 200 | Rome | 1 |
| 500 | Paris | 2 |
| 100 | London | 1 |
| 300 | Paris | 1 |
| 300 | Rome | 2 |
| 400 | London | 2 |
| 400 | Rome | 3 |

PARTITION BY city ORDER BY month

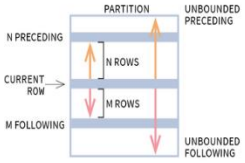
| sold | city | month |
|---|---|---|
| 300 | Paris | 1 |
| 500 | Paris | 2 |
| 200 | Rome | 1 |
| 300 | Rome | 2 |
| 400 | Rome | 3 |
| 100 | London | 1 |
| 400 | London | 2 |

**Default ORDER BY:** with no ORDER BY clause, the order of rows within each partition is arbitrary.

### WINDOW FRAME

is a set of rows that are somehow related to the current row. The window frame is evaluated separately within each partition.

*ROWS | RANGE | GROUPS BETWEEN lower_bound AND upper_bound*




The bounds can be any of the five options:

- UNBOUNDED PRECEDING
- n PRECEDING
- CURRENT ROW
- n FOLLOWING
- UNBOUNDED FOLLOWING

The lower_bound must be BEFORE the upper_bound

ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING

| | city | sold | month |
|---|---|---|---|
| | Paris | 300 | 1 |
| | Rome | 200 | 1 |
| | Paris | 500 | 2 |
| | Rome | 100 | 4 |
| current row → | Paris | 200 | 4 |
| | Paris | 300 | 5 |
| | Rome | 200 | 5 |
| | London | 200 | 5 |
| | London | 100 | 6 |
| | Rome | 300 | 6 |

1 row before the current row and 1 row after the current row

RANGE BETWEEN 1 PRECEDING AND 1 FOLLOWING

| | city | sold | month |
|---|---|---|---|
| | Paris | 300 | 1 |
| | Rome | 200 | 1 |
| | Paris | 500 | 2 |
| | Rome | 100 | 4 |
| current row → | Paris | 200 | 4 |
| | Paris | 300 | 5 |
| | Rome | 200 | 5 |
| | London | 200 | 5 |
| | London | 100 | 6 |
| | Rome | 300 | 6 |

values in the range between 3 and 5 ORDER BY must contain a single expression

GROUPS BETWEEN 1 PRECEDING AND 1 FOLLOWING

| | city | sold | month |
|---|---|---|---|
| | Paris | 300 | 1 |
| | Rome | 200 | 1 |
| | Paris | 500 | 2 |
| | Rome | 100 | 4 |
| current row → | Paris | 200 | 4 |
| | Paris | 300 | 5 |
| | Rome | 200 | 5 |
| | London | 200 | 5 |
| | London | 100 | 6 |
| | Rome | 300 | 6 |

1 group before the current row and 1 group after the current row regardless of the value

As of 2020, GROUPS is only supported in PostgreSQL 11 and up.

### ABBREVIATIONS

| Abbreviation | Meaning |
|---|---|
| UNBOUNDED PRECEDING | BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW |
| n PRECEDING | BETWEEN n PRECEDING AND CURRENT ROW |
| CURRENT ROW | BETWEEN CURRENT ROW AND CURRENT ROW |
| n FOLLOWING | BETWEEN AND CURRENT ROW AND n FOLLOWING |
| UNBOUNDED FOLLOWING | BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING |

### DEFAULT WINDOW FRAME

If ORDER BY is specified, then the frame is RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.

Without ORDER BY, the frame specification is ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.

## Examples of Window Functions

1. **ROW_NUMBER**:
   - Assigns a unique number to each row based on the specified order.
   - Syntax:

```
SELECT
    column1,
    ROW_NUMBER() OVER (PARTITION BY column2 ORDER BY column3)
AS row_num
FROM
    table_name;
```

- Example:

```
SELECT EmployeeID, FirstName, LastName, DepartmentID,
       ROW_NUMBER() OVER (PARTITION BY DepartmentID ORDER BY
EmployeeID) AS RowNum
FROM Employees;
```

2. **RANK**:
   - Assigns a rank to each row within a partition, with gaps for ties.
   - Syntax:

```
SELECT
    column1,
    RANK() OVER (PARTITION BY column2 ORDER BY column3) AS
rank
FROM
    table_name;
```

   - Example:

```
SELECT EmployeeID, FirstName, DepartmentID, Salary,

       RANK() OVER (PARTITION BY DepartmentID ORDER BY Salary
DESC) AS SalaryRank

FROM Employees;
```

3. **DENSE_RANK**:
   - Similar to RANK, but without gaps between ranks.
   - Syntax:

```
SELECT
    column1,
    DENSE_RANK() OVER (PARTITION BY column2 ORDER BY column3)
AS dense_rank
FROM
    table_name;
```

- Example:

```
SELECT EmployeeID, FirstName, DepartmentID, Salary,

 DENSE_RANK() OVER (PARTITION BY DepartmentID ORDER BY
Salary DESC) AS SalaryRank

FROM Employees;
```

### 4. SUM() with Window Function

- You can use SUM() as a window function to calculate a cumulative sum.
- Example:

```
SELECT EmployeeID, FirstName, LastName, Salary,
       SUM(Salary) OVER (ORDER BY EmployeeID) AS
CumulativeSalary
FROM Employees;
```

### 5. LEAD( ) and LAG( )

- LEAD( ) and LAG( ) access data from the subsequent or previous row in the result set.
- Example:

```
SELECT column1,
     LEAD(column2, 1) OVER (ORDER BY column3) AS NextSalary,
     LAG(column2, 1) OVER (ORDER BY column3) AS PreviousSalary
FROM Employees;
```

## Benefits of Using Window Functions

1. **Flexibility**: Can perform calculations across rows without collapsing them into a single result.
2. **Enhanced Analysis:** Allows for complex calculations like ranking, running totals, and moving averages.
3. **Simplified Queries:** Reduces the need for self-joins or subqueries to achieve similar results.

## When to Use Window Functions

1. When you need calculations that consider other rows in the result set.
2. To perform ranking, cumulative sums, moving averages, and similar operations.
3. When working with time series data or ordered data.

## Limitations

1. May have performance implications on large datasets due to the need to process many rows.
2. Complexity in understanding and writing queries, especially with multiple window functions.

**By following this detailed manual, you will develop a comprehensive understanding of advanced query techniques, and how to effectively use these concepts to write powerful SQL queries.**