

Session 6 – Manual

Window Functions

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

Objective:

By the end of this session, participants will be able to:

1. Understand Window Functions:

- Define window functions and explain their use cases.
- Describe the syntax of window functions, including the `OVER` clause, `PARTITION BY`, `ORDER BY`, and frame clauses.

2. Apply Window Functions:

- 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.

3. Combine Query Results:

- Utilize the `UNION` operator to combine the results of multiple `SELECT` statements and understand how it removes duplicates.
- Apply the `INTERSECT` operator to find common rows between multiple `SELECT` statements.
- Use the `EXCEPT` operator to identify differences between two `SELECT` statements.

4. Develop Advanced SQL Queries:

- Integrate window functions with other SQL clauses to create advanced and efficient queries.
- Enhance data retrieval processes by combining results from different queries using set operations.

5. Practical Application and Problem-Solving:

- Solve real-world data problems by applying window functions and data modification techniques.
- Work on hands-on exercises and case studies to reinforce the concepts learned during the session.

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.

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.

Aggregate Functions

Window Functions

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	7. SELECT
2. WHERE	8. DISTINCT
3. GROUP BY	9. UNION/INTERSECT/EXCEPT
4. aggregate functions	10. ORDER BY
5. HAVING	11. OFFSET
6. window functions	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.

PARTITION BY city

month	city	month	city	month	city	month	city
1	Rome	200	1	Paris	300	1	Rome
2	Paris	300	2	Paris	300	2	Rome
3	London	100	3	Rome	200	3	Paris
4	Paris	300	4	Rome	200	4	Paris
5	Rome	300	5	Rome	400	5	Paris
6	London	400	6	London	100	6	Paris
7	Rome	400	7	London	400	7	Paris

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.

PARTITION BY city ORDER BY month

month	city	month	city	month	city	month	city
200	Rome	1	300	Paris	1	200	Paris
300	Paris	2	300	Paris	2	300	Rome
100	London	3	200	Rome	3	300	Rome
300	Paris	4	200	Paris	4	400	Paris
300	Rome	5	200	Rome	5	200	London
400	London	6	100	London	6	200	London
400	Rome	7	400	London	7	400	London

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	month	month
Paris	300	1
Rome	200	1
Paris	300	2
Rome	200	3
Paris	300	4
Rome	200	5
Paris	300	6
London	200	6
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	month	month
Paris	300	1
Rome	200	1
Paris	300	2
Rome	200	3
Paris	300	4
Rome	200	5
Paris	300	6
London	200	6
London	100	6
Rome	300	6

values in the range between 3 and 6
ORDER BY must contain a single expression

GROUPS BETWEEN 1 PRECEDING AND 1 FOLLOWING

city	month	month
Paris	300	1
Rome	200	1
Paris	300	2
Rome	200	3
Paris	300	4
Rome	200	5
Paris	300	6
London	200	6
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 CURRENT ROW AND n FOLLOWING
UNBOUNDED FOLLOWING	BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING

DEFAULT WINDOW FRAME

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.

Syntax of Window Functions

A window function has the following syntax:

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.

Examples of Window Functions

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

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

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

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

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

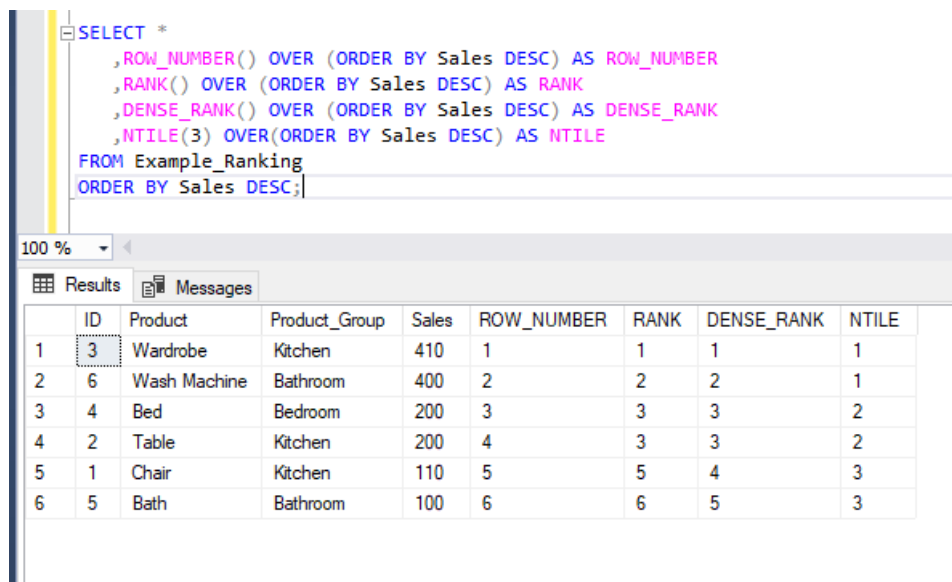
```
SELECT
    column1,
    DENSE_RANK() OVER (PARTITION BY column2 ORDER BY column3) AS
dense_rank
```

```
FROM  
    table_name;
```

Ranking Data

Using RANK

General Example of Rank



The screenshot shows a SQL query editor with a query window and a results window. The query window contains the following SQL code:

```
SELECT *  
    ,ROW_NUMBER() OVER (ORDER BY Sales DESC) AS ROW_NUMBER  
    ,RANK() OVER (ORDER BY Sales DESC) AS RANK  
    ,DENSE_RANK() OVER (ORDER BY Sales DESC) AS DENSE_RANK  
    ,NTILE(3) OVER(ORDER BY Sales DESC) AS NTILE  
FROM Example_Ranking  
ORDER BY Sales DESC;
```

The results window displays the following table:

	ID	Product	Product_Group	Sales	ROW_NUMBER	RANK	DENSE_RANK	NTILE
1	3	Wardrobe	Kitchen	410	1	1	1	1
2	6	Wash Machine	Bathroom	400	2	2	2	1
3	4	Bed	Bedroom	200	3	3	3	2
4	2	Table	Kitchen	200	4	3	3	2
5	1	Chair	Kitchen	110	5	5	4	3
6	5	Bath	Bathroom	100	6	6	5	3

The `RANK` function provides a rank for each row within a partition of a result set, with the same rank assigned to rows with identical values. This can be useful for identifying top performers in groups or similar use cases.

```
SELECT  
    employee_id,  
    department_id,  
    salary,  
    RANK() OVER (PARTITION BY department_id ORDER BY salary DESC) AS rank  
FROM  
    employees;
```

Using DENSE_RANK

`DENSE_RANK` works similarly to `RANK`, but it does not leave gaps in the ranking sequence. Consecutive rows with identical values receive the same rank, and the next rank is incremented by 1.

```
SELECT
    employee_id,
    department_id,
    salary,
    DENSE_RANK() OVER (PARTITION BY department_id ORDER BY salary DESC) AS
dense_rank
FROM employees;
```

Combining Query Results

Using UNION

The `UNION` operator combines the results of two or more `SELECT` statements, removing duplicate rows.

```
SELECT column1, column2 FROM table1
UNION
SELECT column1, column2 FROM table2;
```

Using INTERSECT

The `INTERSECT` operator returns the intersection of two or more `SELECT` statements, i.e., rows that are present in all queries.

```
SELECT column1, column2 FROM table1
INTERSECT
SELECT column1, column2 FROM table2;
```

Using EXCEPT

The `EXCEPT` operator returns the difference between two `SELECT` statements, i.e., rows from the first query that are not in the second query.

```
SELECT column1, column2 FROM table1
EXCEPT
SELECT column1, column2 FROM table2;
```

This manual provides a comprehensive guide to working with window functions in SQL. By mastering these techniques, you can perform sophisticated data analysis and efficiently manage database records.