**Window Functions in SQL**

Window functions in SQL perform calculations across a set of table rows related to the current row. They differ from aggregate functions because they do not collapse the result set into a single row but rather return multiple rows while calculating values based on a defined window of data.

Some commonly used window functions include:

1. RANK()
2. DENSE\_RANK()
3. ROW\_NUMBER()
4. NTILE()
5. LEAD()
6. LAG()

**Examples and Real-Life Scenarios**

**1. RANK()**

Example:

SELECT employee\_id, name, salary,

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

FROM employees;

Explanation:  
This query ranks employees based on their salary in descending order. Employees with the same salary receive the same rank, but the next rank is skipped.

Example Output:

|  |  |  |  |
| --- | --- | --- | --- |
| employee\_id | name | salary | rank |
| 1 | John | 50000 | 1 |
| 2 | Alice | 45000 | 2 |
| 3 | Bob | 45000 | 2 |
| 4 | Charlie | 40000 | 4 |
| 5 | David | 35000 | 5 |

**2. DENSE\_RANK()**

Example:

SELECT employee\_id, name, salary,

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

FROM employees;

Explanation:  
This ranks employees based on their salary in descending order. Unlike RANK(), no ranks are skipped for ties.

Example Output:

|  |  |  |  |
| --- | --- | --- | --- |
| employee\_id | name | salary | dense\_rank |
| 1 | John | 50000 | 1 |
| 2 | Alice | 45000 | 2 |
| 3 | Bob | 45000 | 2 |
| 4 | Charlie | 40000 | 3 |
| 5 | David | 35000 | 4 |

**3. ROW\_NUMBER()**

Example:

SELECT employee\_id, name, salary,

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

FROM employees;

Explanation:  
This assigns a unique row number to each row, regardless of ties. This is useful when you need a unique identifier for each row.

Example Output:

|  |  |  |  |
| --- | --- | --- | --- |
| employee\_id | name | salary | row\_num |
| 1 | John | 50000 | 1 |
| 2 | Alice | 45000 | 2 |
| 3 | Bob | 45000 | 3 |
| 4 | Charlie | 40000 | 4 |
| 5 | David | 35000 | 5 |

**4. NTILE()**

Example:

SELECT employee\_id, name, salary,

NTILE(4) OVER (ORDER BY salary DESC) AS quartile

FROM employees;

Explanation:  
This divides the employees into 4 quartiles based on their salary.

Example Output:

**Expected Output (with 4 quartiles):**

|  |  |  |  |
| --- | --- | --- | --- |
| **employee\_id** | **name** | **salary** | **quartile** |
| 1 | John | 50000 | 1 |
| 2 | Alice | 45000 | 1 |
| 3 | Bob | 45000 | 2 |
| 4 | Charlie | 40000 | 3 |
| 5 | David | 35000 | 4 |

The NTILE(4) function divides the employees into 4 quartiles.

* The first two employees (John and Alice) are placed in **Quartile 1**.
* Bob and Charlie are placed in **Quartile 2 and 3** respectively.
* David is placed in **Quartile 4**.

**5. LEAD()**

Example:

SELECT employee\_id, name, salary,

LEAD(salary, 1) OVER (ORDER BY salary DESC) AS next\_salary

FROM employees;

Explanation:  
This fetches the salary of the next employee in the result set. If the next row doesn't exist, it returns NULL.

Example Output:

| employee\_id | name | salary | next\_salary |
| --- | --- | --- | --- |
| 1 | John | 50000 | 45000 |
| 2 | Alice | 45000 | 45000 |
| 3 | Bob | 45000 | 40000 |
| 4 | Charlie | 40000 | 35000 |
| 5 | David | 35000 | NULL |

**6. LAG()**

Example:

SELECT employee\_id, name, salary,

LAG(salary, 1) OVER (ORDER BY salary DESC) AS previous\_salary

FROM employees;

Explanation:  
This fetches the salary of the previous employee in the result set. If the previous row doesn't exist, it returns NULL.

Example Output:

| employee\_id | name | salary | previous\_salary |
| --- | --- | --- | --- |
| 1 | John | 50000 | NULL |
| 2 | Alice | 45000 | 50000 |
| 3 | Bob | 45000 | 45000 |
| 4 | Charlie | 40000 | 45000 |
| 5 | David | 35000 | 40000 |

Comparison of RANK(), DENSE\_RANK(), ROW\_NUMBER()

|  |  |  |  |
| --- | --- | --- | --- |
| Function | Behavior | Use Case | Real-Life Example |
| RANK() | Assigns ranks, skipping numbers in case of ties. | When you want to rank items, but gaps should be left in ranks when there are ties. | Ranking students based on exam scores (skip ranks in ties). |
| DENSE\_RANK() | Assigns ranks without skipping numbers in case of ties. | When you want ranks to be sequential even when there are ties. | Ranking products in a competition (no gaps in ranks). |
| ROW\_NUMBER() | Assigns a unique number to each row regardless of ties. | When you need a unique identifier for each row, regardless of duplicates or ties. | Assigning a unique ID to each customer order. |

**Importance of Window Functions**

1. **Efficiency in Calculation**: Window functions allow calculations without the need for complex subqueries, making it easier to analyze and summarize data.
2. **Data Comparison**: They allow you to compare values within a specific window (e.g., comparing an employee’s salary with the next or previous employee's salary).
3. **Preserving Data Structure:** Unlike aggregate functions that reduce the result set, window functions preserve the original number of rows, making them useful for row-wise calculations.
4. **Enhanced Analytics:** Window functions enable more advanced analytics like moving averages, ranking, and partitioning without data loss.

provided calculates a 3-day moving average of sales. Let's break down the query:

**Example Query:**

SELECT order\_date, sales,

AVG(sales) OVER (ORDER BY order\_date ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS moving\_avg

FROM sales\_data;

**Explanation:**

**AVG(sales)**: This is the aggregation function that calculates the average of the sales column.

**OVER (ORDER BY order\_date ...)**: This specifies that the moving average should be calculated in the order of order\_date.

**ROWS BETWEEN 2 PRECEDING AND CURRENT ROW**: This defines the window to calculate the moving average over. It includes the current row and the two preceding rows (for a total of 3 rows).

**Output Example:**

Let's assume we have the following sales data in the sales\_data table:

|  |  |
| --- | --- |
| **order\_date** | **sales** |
| 2024-11-01 | 100 |
| 2024-11-02 | 150 |
| 2024-11-03 | 200 |
| 2024-11-04 | 250 |
| 2024-11-05 | 300 |

The query will output something like this:

|  |  |  |
| --- | --- | --- |
| **order\_date** | **sales** | **moving\_avg** |
| 2024-11-01 | 100 | 100 |
| 2024-11-02 | 150 | 125 |
| 2024-11-03 | 200 | 150 |
| 2024-11-04 | 250 | 200 |
| 2024-11-05 | 300 | 250 |

**Explanation of the Output:**

**For 2024-11-01**, the moving average is just 100 since there are no preceding rows.

**For 2024-11-02**, the moving average is (100 + 150) / 2 = 125, as there are two preceding rows (including the current one).

**For 2024-11-03**, the moving average is (100 + 150 + 200) / 3 = 150.

**For 2024-11-04**, the moving average is (150 + 200 + 250) / 3 = 200.

**For 2024-11-05**, the moving average is (200 + 250 + 300) / 3 = 250.

The moving average smooths the data and helps identify trends over time by reducing the effect of short-term fluctuations.

**Summary**

* RANK() is useful for ranking, allowing gaps in ranking for ties.
* DENSE\_RANK() is similar but avoids gaps in ranking.
* ROW\_NUMBER() provides a unique row number for each row, ignoring ties.
* NTILE() divides the result set into equal parts.
* LEAD() and LAG() help with accessing data from the next or previous row.

Window functions are extremely powerful for data analysis, allowing complex calculations while maintaining the structure of the original data, and are essential for more advanced SQL-based analytics.

The PARTITION BY clause in SQL is used to divide the result set into partitions (groups of rows) to apply the window function to each group separately. When used with RANK() and DENSE\_RANK(), it allows you to **rank rows within each partition** rather than across the entire result set.

**Example: RANK() with PARTITION BY**

Let’s say we have a table employee\_sales with sales data for employees in different departments.

|  |  |  |  |
| --- | --- | --- | --- |
| **employee\_id** | **department** | **name** | **sales** |
| 1 | HR | John | 5000 |
| 2 | HR | Alice | 7000 |
| 3 | HR | Bob | 7000 |
| 4 | IT | Charlie | 8000 |
| 5 | IT | David | 6000 |

We want to rank employees within each department based on their sales.

SELECT employee\_id, department, name, sales,

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

FROM employee\_sales;

**Explanation:**

* PARTITION BY department: This groups the rows by department. The RANK() function will assign ranks within each department, starting from 1 for the highest sales in each department.
* ORDER BY sales DESC: This orders the employees within each department by sales in descending order.

**Output:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **employee\_id** | **department** | **name** | **sales** | **rank** |
| 2 | HR | Alice | 7000 | 1 |
| 3 | HR | Bob | 7000 | 1 |
| 1 | HR | John | 5000 | 3 |
| 4 | IT | Charlie | 8000 | 1 |
| 5 | IT | David | 6000 | 2 |

**Explanation of Output:**

* **In the HR department**, Alice and Bob both have the same sales, so they share the same rank (1), and John is ranked 3 because of his lower sales.
* **In the IT department**, Charlie has the highest sales, so he is ranked 1, and David is ranked 2.

**Example: DENSE\_RANK() with PARTITION BY**

Now let’s use DENSE\_RANK() to see how it behaves differently with ties.

SELECT employee\_id, department, name, sales,

DENSE\_RANK() OVER (PARTITION BY department ORDER BY sales DESC) AS dense\_rank

FROM employee\_sales;

**Explanation:**

* DENSE\_RANK() works similarly to RANK(), but it does not skip any rank values when there are ties. So, if two employees have the same sales, they will receive the same rank, but the next rank will not be skipped.

**Output:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **employee\_id** | **department** | **name** | **sales** | **dense\_rank** |
| 2 | HR | Alice | 7000 | 1 |
| 3 | HR | Bob | 7000 | 1 |
| 1 | HR | John | 5000 | 2 |
| 4 | IT | Charlie | 8000 | 1 |
| 5 | IT | David | 6000 | 2 |

**Explanation of Output:**

* **In the HR department**, Alice and Bob both have the same sales and receive rank 1, but John is ranked 2, with no gaps in the ranks.
* **In the IT department**, Charlie is ranked 1, and David is ranked 2, with no skipped ranks.

**Key Difference:**

* **RANK()** skips ranks when there are ties, so if two people are tied for rank 1, the next rank will be 3.
* **DENSE\_RANK()** does not skip any ranks, so after two people are ranked 1, the next rank will be 2.

These window functions with PARTITION BY are particularly useful when you need to perform operations like ranking within groups, categorizing data, or comparing values within subsets of your data.

Here are some real-life examples of how window functions like RANK(), DENSE\_RANK(), ROW\_NUMBER(), LEAD(), LAG(), and NTILE() are used:

**1. RANK() – Ranking Students Based on Exam Scores**

**Scenario**: In a school system, we want to rank students based on their exam scores, but we have ties, meaning two students might score the same.

**Example**:

SELECT student\_id, name, score,

RANK() OVER (ORDER BY score DESC) AS rank

FROM students;

**Real-life interpretation**: This ranks students based on their scores. If two students have the same score, they are tied with the same rank, and the next rank is skipped. For example, if two students are ranked 1, the next student will be ranked 3 (skipping 2).

**2. DENSE\_RANK() – Ranking Products by Popularity**

**Scenario**: A store wants to rank products based on their sales volume. The store doesn't want to skip rank numbers for products with identical sales.

**Example**:

SELECT product\_id, product\_name, sales,

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

FROM products;

**Real-life interpretation**: This ranks products based on sales without skipping any ranks. If two products have the same sales, they get the same rank, and the next rank will continue from the next available rank (i.e., no skipping).

**3. ROW\_NUMBER() – Assigning Unique Order IDs to Customer Purchases**

**Scenario**: A company wants to assign a unique order number to each purchase made by customers, regardless of the order total or customer name.

**Example**:

SELECT order\_id, customer\_id, order\_total,

ROW\_NUMBER() OVER (ORDER BY order\_date) AS unique\_order\_number

FROM orders;

**Real-life interpretation**: This assigns a unique order number to each purchase, regardless of ties, so each purchase gets its own unique row number.

**4. LEAD() – Predicting Future Values (Next Purchase Date)**

**Scenario**: An e-commerce company wants to know the next purchase date for a customer to predict when a customer might return to buy again.

**Example**:

SELECT customer\_id, purchase\_date,

LEAD(purchase\_date, 1) OVER (PARTITION BY customer\_id ORDER BY purchase\_date) AS next\_purchase\_date

FROM purchases;

**Real-life interpretation**: This shows the next purchase date for each customer. For instance, if a customer made a purchase on January 1, this function will show their next purchase date, which helps in understanding customer behavior and predicting future purchases.

**5. LAG() – Comparing Current Month’s Sales to Previous Month’s**

**Scenario**: A sales team wants to compare their current month’s sales with the previous month to understand their growth or decline.

**Example**:

SELECT sales\_month, sales\_amount,

LAG(sales\_amount, 1) OVER (ORDER BY sales\_month) AS previous\_month\_sales

FROM monthly\_sales;

**Real-life interpretation**: This shows the sales amount for the current month and compares it with the sales amount of the previous month. It helps in analyzing trends, such as whether the business is growing or declining.

**6. NTILE() – Dividing Employees into Performance Quartiles**

**Scenario**: A company wants to divide its employees into four quartiles based on performance scores for performance reviews.

**Example**:

SELECT employee\_id, name, performance\_score,

NTILE(4) OVER (ORDER BY performance\_score DESC) AS quartile

FROM employees;

**Real-life interpretation**: This divides employees into four performance quartiles. The first quartile would contain the top-performing employees, while the fourth quartile would contain the lowest-performing employees. This can be useful for targeted interventions or rewards.

**7. SUM() OVER() – Running Total of Sales**

**Scenario**: A manager wants to track a running total of sales to see how the business is performing over time.

**Example**:

SELECT sales\_date, daily\_sales,

SUM(daily\_sales) OVER (ORDER BY sales\_date) AS running\_total\_sales

FROM sales\_data;

**Real-life interpretation**: This calculates the running total of sales over time. As each new day’s sales are added, the running total keeps increasing, giving a sense of cumulative sales progress.

**8. AVG() OVER() – Calculating Moving Averages for Stock Prices**

**Scenario**: A financial analyst wants to calculate a 5-day moving average for stock prices to smooth out short-term fluctuations.

**Example**:

SELECT stock\_date, stock\_price,

AVG(stock\_price) OVER (ORDER BY stock\_date ROWS BETWEEN 4 PRECEDING AND CURRENT ROW) AS moving\_avg

FROM stock\_prices;

**Real-life interpretation**: This calculates the average stock price over the last 5 days. By using a moving average, analysts can observe long-term trends in stock prices without the noise of short-term fluctuations.

**Key Real-life Uses and Benefits of Window Functions:**

|  |  |  |
| --- | --- | --- |
| Window Function | Real-life Use | Benefit |
| RANK() | Ranking students, products, or employees based on scores/sales | Helps rank tied data while skipping ranks for clarity |
| DENSE\_RANK() | Ranking products without skipping ranks for tied values | Ensures ranks are continuous, even with ties |
| ROW\_NUMBER() | Assigning unique IDs to records such as orders or tickets | Provides a unique identifier for each row |
| LEAD() | Predicting next event (e.g., next customer purchase) | Helps in forecasting or trend analysis |
| LAG() | Comparing current data with previous data (e.g., sales growth) | Enables trend analysis over time |
| NTILE() | Dividing data into quartiles, deciles, etc. (e.g., employee performance) | Divides data into parts for easier comparison |
| SUM() OVER() | Calculating running totals (e.g., total sales per day) | Tracks cumulative progress over time |
| AVG() OVER() | Calculating moving averages (e.g., stock price averages) | Smoothens out fluctuations for trend analysis |

These examples show how window functions can be applied in various real-world scenarios to solve complex problems and analyze data in meaningful ways.

Here are some multiple-choice questions (MCQs) on window functions in SQL:

**1. What does the RANK() window function do in SQL?**

A) It assigns a unique rank to each row, skipping ranks for ties.  
B) It assigns a unique rank to each row, with no ranks skipped for ties.  
C) It assigns row numbers to each row without regard to ties.  
D) It calculates the average of the values in the specified window.

**Answer**: A) It assigns a unique rank to each row, skipping ranks for ties.

**2. Which of the following window functions gives a unique number to each row in the result set, even when there are ties?**

A) RANK()  
B) DENSE\_RANK()  
C) ROW\_NUMBER()  
D) NTILE()

**Answer**: C) ROW\_NUMBER()

**3. What is the purpose of the LEAD() function in SQL?**

A) To compare the current row's value with the next row's value.  
B) To compare the current row's value with the previous row's value.  
C) To assign row numbers to each row in a result set.  
D) To divide rows into equal groups based on a specified number.

**Answer**: A) To compare the current row's value with the next row's value.

**4. Which window function would you use to divide the data into a specified number of groups, such as quartiles?**

A) RANK()  
B) DENSE\_RANK()  
C) NTILE()  
D) ROW\_NUMBER()

**Answer**: C) NTILE()

**5. If you want to assign ranks to employees based on their salary in descending order, and you want no ranks skipped even if two employees have the same salary, which window function would you use?**

A) RANK()  
B) ROW\_NUMBER()  
C) DENSE\_RANK()  
D) LEAD()

**Answer**: C) DENSE\_RANK()

**6. What does the ROWS BETWEEN clause do in a window function like AVG()?**

A) It limits the rows to the first few records.  
B) It specifies the number of rows to include in the window, such as a moving average.  
C) It orders the rows in ascending order.  
D) It applies a filter condition to the result set.

**Answer**: B) It specifies the number of rows to include in the window, such as a moving average.

**7. Which window function would you use to get the previous row's value in a dataset?**

A) LEAD()  
B) LAG()  
C) NTILE()  
D) RANK()

**Answer**: B) LAG()

**8. How would you use the NTILE() function to divide a dataset into 5 equal parts?**

A) NTILE(5) OVER (ORDER BY salary)  
B) NTILE(4) OVER (ORDER BY salary)  
C) NTILE(10) OVER (ORDER BY salary)  
D) NTILE(3) OVER (ORDER BY salary)

**Answer**: A) NTILE(5) OVER (ORDER BY salary)

**9. In SQL, what is the difference between RANK() and DENSE\_RANK()?**

A) RANK() assigns ranks without skipping, while DENSE\_RANK() skips ranks in case of ties.  
B) RANK() skips ranks in case of ties, while DENSE\_RANK() assigns consecutive ranks.  
C) There is no difference between the two.  
D) RANK() only works with numeric data, while DENSE\_RANK() works with text data.

**Answer**: B) RANK() skips ranks in case of ties, while DENSE\_RANK() assigns consecutive ranks.

**10. Which window function would you use to calculate a moving average over a specified range of rows?**

A) AVG() OVER (ORDER BY column\_name ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)  
B) SUM() OVER (ORDER BY column\_name ROWS BETWEEN 1 FOLLOWING AND CURRENT ROW)  
C) ROW\_NUMBER() OVER (ORDER BY column\_name)  
D) LEAD() OVER (ORDER BY column\_name)

**Answer**: A) AVG() OVER (ORDER BY column\_name ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)