Mastek
Trust. Value. Velocity

ORACLE SQL

Analytical Functions

FEBRUARY 12, 2024

Contents

Analytical Functions

Analytical functions in Oracle SQL are a set of powerful tools used for performing complex calculations across a set of rows returned by a query. They allow you to compute aggregated values such as moving averages, cumulative sums, rankings, and other statistical calculations without needing to use subqueries or self-joins.

Here are some key characteristics of analytical functions:

1. **Operate on a group of rows:** Analytical functions compute values across a set of rows defined by the query's `PARTITION BY` clause. This allows you to perform calculations within specific groups of data, such as partitions or windows.

2. **Compute values based on order:** Analytical functions often require an ordering of rows to be meaningful. This is achieved using the `ORDER BY` clause within the function's windowing clause.

3. Do not affect result set: Unlike aggregate functions like `SUM()` or `AVG()`, analytical functions do not collapse multiple rows into a single row. Instead, they add new columns to the result set containing the computed values.

4. Commonly used for reporting and analysis: Analytical functions are commonly used in business intelligence, reporting, and data analysis tasks where complex calculations are required to gain insights into data distributions, trends, and patterns.

Examples of analytical functions in Oracle SQL include `RANK()`, `DENSE_RANK()`, `ROW_NUMBER()`, `SUM() OVER()`, `AVG() OVER()`, `LEAD()`, `LAG()`, `NTILE()`, `FIRST_VALUE()`, `LAST_VALUE()`, and `CUME_DIST()`.

Overall, analytical functions enhance the capabilities of SQL queries by providing a concise and efficient way to perform complex calculations and analysis on result sets. They are invaluable tools for data professionals working with large datasets or needing to perform sophisticated computations within SQL queries.

Rank()

Why use RANK()?

Purpose: The RANK() function is used to assign a rank to each row in a result set based on the values of specified columns. It's valuable for scenarios where you need to determine the relative position of rows according to certain criteria, such as ordering employees by salary or sales by revenue.

What is RANK()?

Functionality: RANK() assigns a unique rank to each distinct row in the result set, handling ties by leaving gaps in the ranking sequence. For example, if two rows tie for the first position, the next rank will be the third. This behavior is different from DENSE_RANK(), which assigns consecutive ranks without gaps.

How to use RANK()?

Syntax:

```
RANK() OVER (PARTITION BY column1, column2, ... ORDER BY expression)
```

PARTITION BY: This optional clause divides the result set into partitions, and ranking is performed independently within each partition.

ORDER BY: Specifies the column or expression based on which the ranking is determined. The result set is ordered accordingly.

We want to rank employees by their salary within each department using the RANK() function.

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

Explanation of Execution:

Partitioning: The PARTITION BY department_id clause divides the result set into partitions based on the department_id. This means that ranking will be done separately for each department.

Ordering: The ORDER BY salary DESC clause orders the rows within each partition by salary in descending order. This determines the ranking sequence for employees within each department.

Ranking: The RANK() function then assigns a rank to each row within its partition based on the order specified. Ties (employees with the same salary within a department) will receive the same rank, and subsequent ranks will be skipped accordingly.

This query will return the employee_id, employee_name, department_id, salary, and the salary_rank column, which represents the rank of each employee's salary within their respective department.

Summary:

Why use Rank(): To determine the relative position of rows based on specified criteria.

What is Rank(): It assigns a unique rank to each row in the result set, handling ties with a gap in the ranking sequence.

How to use Rank(): Specify partitions with PARTITION BY, ordering with ORDER BY, and understand that ties receive the same rank.

Dense_Rank()

Why use DENSE_RANK()?

Purpose: The DENSE_RANK() function is used to assign a rank to each row in a result set based on the values of specified columns. It is similar to RANK(), but it does not leave gaps in the ranking sequence. This function is useful when you want consecutive ranks without any gaps, particularly when handling tied values.

What is DENSE_RANK()?

Functionality: DENSE_RANK() assigns a unique rank to each distinct row in the result set, handling ties by assigning consecutive ranks without any gaps. For instance, if two rows tie for the first position, both will be assigned a rank of 1, and the next row will receive a rank of 2.

How to use DENSE_RANK()?

```
DENSE_RANK() OVER (PARTITION BY column1, column2, ... ORDER BY expression)
```

PARTITION BY: This optional clause divides the result set into partitions, and ranking is performed independently within each partition.

ORDER BY: Specifies the column or expression based on which the ranking is determined. The result set is ordered accordingly.

We want to rank employees by their salary within each department using the DENSE_RANK() function.

```
SELECT
    employee_id,
    employee_name,
    department_id,
    salary,
    DENSE_RANK() OVER (PARTITION BY department_id ORDER BY salary DESC) AS salary_dense_rank
FROM
```

```
employees;
```

Explanation of Execution:

Partitioning: The PARTITION BY department_id clause divides the result set into partitions based on the department_id. This means that ranking will be done separately for each department.

Ordering: The ORDER BY salary DESC clause orders the rows within each partition by salary in descending order. This determines the ranking sequence for employees within each department.

Ranking: The DENSE_RANK() function then assigns a rank to each row within its partition based on the order specified. Ties (employees with the same salary within a department) will receive the same rank, and subsequent ranks will be assigned consecutively without any gaps.

This query will return the employee_id, employee_name, department_id, salary, and the salary_dense_rank column, which represents the dense rank of each employee's salary within their respective department.

Summary:

Why use Dense_Rank(): To determine the relative position of rows based on specified criteria, ensuring consecutive ranks without any gaps.

What is Dense_Rank(): It assigns a unique rank to each row in the result set, handling ties by assigning consecutive ranks without any gaps.

How to use Dense_Rank(): Specify partitions with PARTITION BY, ordering with ORDER BY, and understand that ties receive the same rank but subsequent ranks are assigned consecutively without gaps.

Row_Number()

Why use ROW_NUMBER()?

Purpose: The ROW_NUMBER() function is used to assign a unique sequential integer to each row in the result set. It's valuable when you need to enumerate rows without considering any specific order or grouping.

What is ROW_NUMBER()?

Functionality: ROW_NUMBER() assigns a unique sequential integer to each row in the result set, starting from 1 and incrementing by 1 for each subsequent row. Unlike RANK() and DENSE_RANK(), it does not handle ties; each row is assigned a distinct number.

How to use ROW_NUMBER()?

```
ROW_NUMBER() OVER (ORDER BY expression)

ORDER BY: This mandatory clause specifies the column or expression based on which the rows are ordered. The result set is then numbered sequentially according to this order.
```

We want to assign a unique sequential number to each employee in the HR schema.

```
SELECT
   employee_id,
   employee_name,
   department_id,
   ROW_NUMBER() OVER (ORDER BY employee_id) AS row_num
FROM
   employees;
```

Explanation of Execution:

Ordering: The ORDER BY employee_id clause orders the rows in the result set based on the employee_id. This determines the sequence in which the rows will be numbered.

Numbering: The ROW_NUMBER() function then assigns a sequential number to each row in the ordered result set. The numbering starts from 1 and increments by 1 for each subsequent row.

This query will return the employee_id, employee_name, department_id, and the row_num column, which represents the sequential number assigned to each row.

Summary:

Why use Row_Number(): To assign a unique sequential integer to each row in the result set.

What is Row_Number(): It assigns a sequential number to each row, starting from 1 and incrementing by 1 for each subsequent row.

How to use Row_Number(): Specify the ordering with ORDER BY, and understand that each row is assigned a distinct number sequentially.

Sum() Over()

Why use SUM() OVER()?

Purpose: The SUM() OVER() function is used to calculate a cumulative sum over a set of rows defined by the query. It's valuable when you need to compute running totals, such as accumulating sales revenue or calculating year-to-date expenses.

What is SUM() OVER()?

Functionality: SUM() OVER() calculates a cumulative sum of a specified column or expression over a window of rows defined by the OVER() clause. It computes the sum for the current row and all preceding rows within the window.

How to use SUM() OVER()?

```
SUM(column_name) OVER (PARTITION BY partition_column ORDER BY order_column)

column_name: Specifies the column or expression to be summed.

PARTITION BY: Optional clause that divides the result set into partitions. The sum is calculated separately for each partition.
```

ORDER BY: Specifies the column or expression based on which the rows are ordered within each partition.

We want to calculate the running total of salaries within each department in the HR schema.

```
SELECT
    employee_id,
    employee_name,
    department_id,
    salary,
    SUM(salary) OVER (PARTITION BY department_id ORDER BY employee_id) AS
department_running_total
FROM
    employees;
```

Explanation of Execution:

Partitioning: The PARTITION BY department_id clause divides the result set into partitions based on the department_id. This means that the running total will be calculated separately for each department.

Ordering: The ORDER BY employee_id clause orders the rows within each partition by employee_id. This determines the sequence in which the rows will be processed for calculating the running total.

Calculating Running Total: The SUM(salary) OVER() function then calculates the cumulative sum of salaries within each partition, considering the order specified. It computes the sum for the current row and all preceding rows within the partition.

This query will return the employee_id, employee_name, department_id, salary, and the department_running_total column, which represents the running total of salaries within each department.

Summary:

Why use Sum() Over(): To calculate a cumulative sum over a set of rows, such as computing running totals or cumulative aggregates.

What is Sum() Over(): It calculates a cumulative sum of a specified column or expression over a window of rows defined by the OVER() clause.

How to use Sum() Over(): Specify the column to be summed, partitions with PARTITION BY, and ordering with ORDER BY.

Avg() Over()

Why use AVG() OVER()?

Purpose: The AVG() OVER() function is used to calculate the average value of a column or expression over a window of rows defined by the query. It's beneficial when you need to compute the average over a subset of data while retaining the granularity of the original rows.

What is AVG() OVER()?

Functionality: AVG() OVER() calculates the average value of a specified column or expression over a window of rows defined by the OVER() clause. It computes the average for the current row and all preceding rows within the window.

How to use AVG() OVER()?

```
AVG(column_name) OVER (PARTITION BY partition_column ORDER BY order_column)

  column_name: Specifies the column or expression for which the average is to be calculated.

  PARTITION BY: Optional clause that divides the result set into partitions. The average is calculated separately for each partition.

  ORDER BY: Specifies the column or expression based on which the rows are ordered within each partition.
```

We want to calculate the average salary within each department in the HR schema.

```
SELECT
    employee_id,
    employee_name,
    department_id,
    salary,
    AVG(salary) OVER (PARTITION BY department_id) AS department_average_salary
FROM
    employees;
```

Explanation of Execution:

Partitioning: The PARTITION BY department_id clause divides the result set into partitions based on the department_id. This means that the average salary will be calculated separately for each department.

Calculating Average: The AVG(salary) OVER() function then calculates the average salary over each partition. It computes the average for the current row and all other rows within the same department.

This query will return the employee_id, employee_name, department_id, salary, and the department_average_salary column, which represents the average salary within each department.

Summary:

Why use Avg() Over(): To calculate the average value of a column or expression over a window of rows, such as computing average sales or average scores within groups.

What is Avg() Over(): It calculates the average value of a specified column or expression over a window of rows defined by the OVER() clause.

How to use Avg() Over(): Specify the column for which the average is to be calculated, partitions with PARTITION BY, and understand that the average is computed for each partition.

Lead()

Why use LEAD()?

Purpose: The LEAD() function is used to access data from a subsequent row in the result set. It's beneficial when you need to compare data across adjacent rows or access values from the next row for each row in the result set.

What is LEAD()?

Functionality: LEAD() retrieves the value from the next row in the result set based on a specified column or expression. It provides access to subsequent data without the need for self-joins or subqueries.

How to use LEAD()?

```
LEAD(column_name, offset, default_value) OVER (ORDER BY order_column)

  column_name: Specifies the column or expression whose value from the next row is to be retrieved.

  offset: Optional parameter that specifies the number of rows forward from the current row. The default value is 1.

  default_value: Optional parameter that specifies the default value to return if the next row does not exist or if the offset goes beyond the result set.
```

We want to retrieve the name of the next employee (ordered by employee_id) for each employee in the HR schema.

```
SELECT
    employee_id,
    employee_name,
    LEAD(employee_name) OVER (ORDER BY employee_id) AS next_employee_name
FROM
    employees;
```

Explanation of Execution:

Ordering: The ORDER BY employee_id clause orders the rows in the result set based on the employee_id. This determines the sequence in which the rows will be processed for accessing the next employee's name.

Accessing Next Value: The LEAD(employee_name) function then retrieves the name of the next employee for each row in the result set, based on the specified order. If the current row is the last row, or if the offset goes beyond the result set, the function returns null.

This query will return the employee_id, employee_name, and the next_employee_name column, which contains the name of the next employee for each row.

Summary:

Why use Lead(): To access data from the next row in the result set, allowing for comparisons or calculations across adjacent rows.

What is Lead(): It retrieves the value from the next row based on a specified column or expression, providing access to subsequent data without the need for self-joins or subqueries.

How to use Lead(): Specify the column for which the next value is to be retrieved, ordering with ORDER BY, and understand that the function returns null if the next row does not exist.

NTile()

Why use NTILE()?

Purpose: The NTILE() function is used to divide the result set into a specified number of approximately equal-sized groups or "tiles." It's useful when you need to distribute data evenly across a predetermined number of partitions.

What is NTILE()?

Functionality: NTILE() assigns a tile number to each row in the result set, based on the specified number of tiles. It divides the result set into roughly equal-sized groups, ensuring that each group has approximately the same number of rows.

How to use NTILE()?

```
NTILE(number_of_tiles) OVER (ORDER BY order_column)

number_of_tiles: Specifies the desired number of tiles into which the result set should be divided.

ORDER BY: Specifies the column or expression based on which the rows are ordered. This determines the sequence in which the rows will be assigned to tiles.
```

We want to divide employees into three equal-sized groups based on their salaries in the HR schema.

```
SELECT
    employee_id,
    employee_name,
    salary,
    NTILE(3) OVER (ORDER BY salary DESC) AS salary_group
FROM
    employees;
```

Explanation of Execution:

Ordering: The ORDER BY salary DESC clause orders the rows in the result set based on the salary in descending order. This determines the sequence in which the rows will be assigned to tiles.

Dividing into Tiles: The NTILE(3) function then divides the result set into three approximately equal-sized groups based on salary. It assigns a tile number to each row, ensuring that each group has roughly the same number of rows.

This query will return the employee_id, employee_name, salary, and the salary_group column, which represents the group number to which each employee belongs based on their salary.

Summary:

Why use NTile(): To divide the result set into a specified number of approximately equal-sized groups or "tiles," ensuring even distribution of data.

What is NTile(): It assigns a tile number to each row in the result set based on the specified number of tiles, ensuring that each group has roughly the same number of rows.

How to use NTile(): Specify the desired number of tiles, ordering with ORDER BY, and understand that each row is assigned a tile number indicating its group membership.

First_Value()

Why use FIRST_VALUE()?

Purpose: The FIRST_VALUE() function is used to retrieve the value of a specified column from the first row in an ordered partition of the result set. It's helpful when you need to access the first value within a group or partition, such as identifying the earliest date or the lowest salary.

What is FIRST_VALUE()?

Functionality: FIRST_VALUE() retrieves the value of a specified column from the first row in an ordered partition of the result set. It returns the same value for all rows within the partition.

How to use FIRST_VALUE()?

```
FIRST_VALUE(column_name) OVER (PARTITION BY partition_column ORDER BY order_column)

  column_name: Specifies the column whose value from the first row in the partition is to be retrieved.

  PARTITION BY: Optional clause that divides the result set into partitions. The first value is retrieved separately for each partition.

  ORDER BY: Specifies the column or expression based on which the rows are ordered within each partition.
```

We want to retrieve the name of the employee with the lowest salary within each department in the HR schema.

```
SELECT
    employee_id,
    employee_name,
    department_id,
    salary,
    FIRST_VALUE(employee_name) OVER (PARTITION BY department_id ORDER BY salary) AS lowest_salary_employee
FROM
    employees;
```

Explanation of Execution:

Partitioning: The PARTITION BY department_id clause divides the result set into partitions based on the department_id. This means that the first value will be retrieved separately for each department.

Ordering: The ORDER BY salary clause orders the rows within each partition by salary. This determines the sequence in which the rows will be processed for retrieving the first value.

Retrieving First Value: The FIRST_VALUE(employee_name) function then retrieves the name of the employee with the lowest salary within each department. It returns the same value (the name of the employee with the lowest salary) for all rows within the partition.

This query will return the employee_id, employee_name, department_id, salary, and the lowest_salary_employee column, which contains the name of the employee with the lowest salary within each department.

Summary:

Why use First_Value(): To retrieve the value of a specified column from the first row in an ordered partition of the result set, such as identifying the earliest date or the lowest salary.

What is First_Value(): It retrieves the value of a specified column from the first row in an ordered partition of the result set, returning the same value for all rows within the partition.

How to use First_Value(): Specify the column for which the first value is to be retrieved, partitions with PARTITION BY, and ordering with ORDER BY.

Last_value()

Why use LAST_VALUE()?

Purpose: The LAST_VALUE() function is used to retrieve the value of a specified column from the last row in an ordered partition of the result set. It's useful when you need to access the last value within a group or partition, such as identifying the latest date or the highest salary.

What is LAST_VALUE()?

Functionality: LAST_VALUE() retrieves the value of a specified column from the last row in an ordered partition of the result set. It returns the same value for all rows within the partition.

How to use LAST_VALUE()?

```
LAST_VALUE(column_name) OVER (PARTITION BY partition_column ORDER BY order_column ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
```

column_name: Specifies the column whose value from the last row in the partition is to be retrieved.

PARTITION BY: Optional clause that divides the result set into partitions. The last value is retrieved separately for each partition.

ORDER BY: Specifies the column or expression based on which the rows are ordered within each partition.

ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING: This clause ensures that the window for the function includes all rows within the partition.

We want to retrieve the name of the employee with the highest salary within each department in the HR schema.

```
SELECT
    employee_id,
    employee_name,
    department_id,
    salary,
    LAST_VALUE(employee_name) OVER (PARTITION BY department_id ORDER BY salary ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS highest_salary_employee
FROM
    employees;
```

Explanation of Execution:

Partitioning: The PARTITION BY department_id clause divides the result set into partitions based on the department_id. This means that the last value will be retrieved separately for each department.

Ordering: The ORDER BY salary clause orders the rows within each partition by salary. This determines the sequence in which the rows will be processed for retrieving the last value.

Retrieving Last Value: The LAST_VALUE(employee_name) function then retrieves the name of the employee with the highest salary within each department. It returns the same value (the name of the employee with the highest salary) for all rows within the partition.

This query will return the employee_id, employee_name, department_id, salary, and the highest_salary_employee column, which contains the name of the employee with the highest salary within each department.

Summary:

Why use Last_Value(): To retrieve the value of a specified column from the last row in an ordered partition of the result set, such as identifying the latest date or the highest salary.

What is Last_Value(): It retrieves the value of a specified column from the last row in an ordered partition of the result set, returning the same value for all rows within the partition.

How to use Last_Value(): Specify the column for which the last value is to be retrieved, partitions with PARTITION BY, ordering with ORDER BY, and ensuring the window includes all rows within the partition.

Cume_Dist()

Why use CUME_DIST()?

Purpose: The CUME_DIST() function is used to calculate the cumulative distribution of a value within a group of values. It's valuable when you need to determine the relative position of a value within a dataset compared to others.

What is CUME_DIST()?

Functionality: CUME_DIST() calculates the cumulative distribution of a value within a group. It returns the proportion of rows that are less than or equal to the current row's value. Essentially, it indicates the percentile rank of each row's value within the group.

How to use CUME_DIST()?

```
CUME_DIST() OVER (ORDER BY column_name)
```

ORDER BY: Specifies the column or expression based on which the rows are ordered. This determines the sequence in which the rows will be processed for calculating the cumulative distribution.

We want to calculate the cumulative distribution of salaries within each department in the HR schema.

```
SELECT
    employee_id,
    employee_name,
    department_id,
    salary,
    CUME_DIST() OVER (PARTITION BY department_id ORDER BY salary) AS salary_cume_dist
FROM
    employees;
```

Explanation of Execution:

Partitioning: The PARTITION BY department_id clause divides the result set into partitions based on the department_id. This means that the cumulative distribution will be calculated separately for each department.

Ordering: The ORDER BY salary clause orders the rows within each partition by salary. This determines the sequence in which the rows will be processed for calculating the cumulative distribution.

Calculating Cumulative Distribution: The CUME_DIST() function then calculates the cumulative distribution of salaries within each department. It returns the proportion of rows with salaries less than or equal to the current row's salary, indicating the percentile rank of each row's salary within the department.

This query will return the employee_id, employee_name, department_id, salary, and the salary_cume_dist column, which contains the cumulative distribution of salaries within each department.

Summary:

Why use Cume_Dist(): To calculate the cumulative distribution of a value within a group, indicating the percentile rank of each row's value compared to others.

What is Cume_Dist(): It calculates the cumulative distribution of a value within a group, returning the proportion of rows that are less than or equal to the current row's value.

How to use Cume_Dist(): Specify ordering with ORDER BY, and understand that the function calculates the cumulative distribution within each group or partition.

PERCENT_RANK()

Why use PERCENT_RANK()?

Purpose: The PERCENT_RANK() function is used to calculate the relative rank of a value within a group, expressed as a percentage. It's useful when you need to understand the relative position of a value compared to others in a dataset.

What is PERCENT_RANK()?

Functionality: PERCENT_RANK() calculates the relative rank of a value within a group, expressed as a percentage. It indicates the percentile rank of each row's value compared to others within the group.

How to use PERCENT_RANK()?

```
PERCENT_RANK() OVER (ORDER BY column_name)
```

ORDER BY: Specifies the column or expression based on which the rows are ordered. This determines the sequence in which the rows will be processed for calculating the percentile rank.

We want to calculate the percent rank of salaries within each department in the HR schema.

```
SELECT
    employee_id,
    employee_name,
    department_id,
    salary,
    PERCENT_RANK() OVER (PARTITION BY department_id ORDER BY salary) AS salary_percent_rank
```

```
FROM
    employees;
```

Explanation of Execution:

Partitioning: The PARTITION BY department_id clause divides the result set into partitions based on the department_id. This means that the percent rank will be calculated separately for each department.

Ordering: The ORDER BY salary clause orders the rows within each partition by salary. This determines the sequence in which the rows will be processed for calculating the percent rank.

Calculating Percent Rank: The PERCENT_RANK() function then calculates the percent rank of salaries within each department. It returns a value between 0 and 1, indicating the percentile rank of each row's salary within the department.

This query will return the employee_id, employee_name, department_id, salary, and the salary_percent_rank column, which contains the percent rank of salaries within each department.

Summary:

Why use Percent_Rank(): To calculate the relative rank of a value within a group, expressed as a percentage, indicating the percentile rank of each row's value compared to others.

What is Percent_Rank(): It calculates the relative rank of a value within a group, expressed as a percentage, returning a value between 0 and 1.

How to use Percent_Rank(): Specify ordering with ORDER BY, and understand that the function calculates the percent rank within each group or partition.

NTH_VALUE

Why use NTH_VALUE()?

Purpose: The NTH_VALUE() function is used to retrieve the value of a specified column from the nth row in an ordered partition of the result set. It's useful when you need to access the value of a specific row within a group or partition.

What is NTH_VALUE()?

Functionality: NTH_VALUE() retrieves the value of a specified column from the nth row in an ordered partition of the result set. It returns the same value for all rows within the partition.

How to use NTH_VALUE()?

```
NTH_VALUE(column_name, n) OVER (PARTITION BY partition_column ORDER BY order_column)
```

column_name: Specifies the column whose value from the nth row in the partition is to be retrieved.

n: Specifies the position of the row from which to retrieve the value.

PARTITION BY: Optional clause that divides the result set into partitions. The value is retrieved separately for each partition.

ORDER BY: Specifies the column or expression based on which the rows are ordered within each partition.

We want to retrieve the name of the employee with the second-highest salary within each department in the HR schema.

```
SELECT
    employee_id,
    employee_name,
    department_id,
    salary,
    NTH_VALUE(employee_name, 2) OVER (PARTITION BY department_id ORDER BY salary DESC) AS second_highest_salary_employee
FROM
    employees;
```

Explanation of Execution:

Partitioning: The PARTITION BY department_id clause divides the result set into partitions based on the department_id. This means that the value will be retrieved separately for each department.

Ordering: The ORDER BY salary DESC clause orders the rows within each partition by salary in descending order. This determines the sequence in which the rows will be processed for retrieving the value.

Retrieving Nth Value: The NTH_VALUE(employee_name, 2) function then retrieves the name of the employee with the second-highest salary within each department. It returns the same value (the name of the employee with the second-highest salary) for all rows within the partition.

This query will return the employee_id, employee_name, department_id, salary, and the second_highest_salary_employee column, which contains the name of the employee with the second-highest salary within each department.

Summary:

Why use Nth_Value(): To retrieve the value of a specified column from the nth row in an ordered partition of the result set, such as accessing the value of a specific row within a group or partition.

What is Nth_Value(): It retrieves the value of a specified column from the nth row in an ordered partition of the result set, returning the same value for all rows within the partition.

How to use Nth_Value(): Specify the column and the position of the row for which the value is to be retrieved, partitions with PARTITION BY, and ordering with ORDER BY.