Window functions

Window functions in SQL

Types of window functions

> Let’s have a closer look at some specific examples of window functions and how to use them.

Aggregate

- `AVG()`
- `MAX()`
- `MIN()`
- `SUM()`
- `COUNT()`

Ranking

- `ROW_NUMBER`
- `RANK()`
- `DENSE_RANK()`
- `NTILE()`

Analytical

- `LAG()`
- `LEAD()`
- `FIRST_VALUE()`
- `LAST_VALUE()`

Data overview

We will use the following table called `Employee` that contains information about employees in a company located in South Africa. We assume the database is selected, so we don't specify it.

Start_date	Department	Province	First_name	Gender	Salary
2015-06-01	Finance	Gauteng	Lily	Female	35760
2020-08-01	Marketing	Western_Cape	Gabriel	Male	30500
2022-03-01	Data_analytics	Free_State	Maryam	Female	46200
2022-07-15	Marketing	Gauteng	Sophia	Female	36900
2019-05-01	Data_analytics	Western_Cape	Alex	Male	36200
2012-01-01	Finance	Free_State	Martha	Female	48500
2014-05-01	Finance	Western_Cape	Joshua	Male	35760
2017-06-15	Data_analytics	Gauteng	Emily	Female	37800
2016-01-01	Marketing	Western_Cape	David	Male	31000

Aggregate window functions

> Suppose we want to see the **minimum salary** in each **department**. We can use `MIN()` as a window function and return the minimum value to each row.

Query

```
SELECT
   Department,
   First_name,
   Salary,
   MIN(Salary) OVER (
     PARTITION BY Department) AS Min_salary
FROM
   Employee
ORDER BY
   Department;
```

Using `ORDER BY` here sorts the result set by **department**. This has **no effect** on how the **window function** works.

Aggregate window functions

Department	First_name	Salary	Min_salary
Data_analytics	Maryam	46200	36200
Data_analytics	Alex	36200	36200
Data_analytics	Emily	37800	36200
Finance	Joshua	35760	35760
Finance	Lily	35760	35760
Finance	Martha	48500	35760
Marketing	Sophia	36900	30500
Marketing	David	31000	30500
Marketing	Gabriel	30500	30500

The lowest salary in each department is returned to the correct rows.

The minimum value resets for `Finance` and `Marketing` since **each department** is a **different partition**/window.

`MIN()`, `MAX()`, `AVG()`, `SUM()`, and `COUNT()` behave in a similar way when used as window functions.

Aggregate window functions

Suppose we want to see the **total salary budget per department** as a **running total**. Using `SUM()` as a window function, we can add up `Salary` in each partition separately. Adding `ORDER BY` will give a **running total** in each row.

Query

```
SELECT
   Department,
   First_name,
   Salary,
   SUM(Salary) OVER (
     PARTITION BY Department
     ORDER BY Date_Started) AS Dept_salary_budget
FROM
   Employee
ORDER BY
   Department;
```

Adding `ORDER BY` here enables the running sum calculation by date. All aggregate functions can be modified like this.

Window functions in SQL

Aggregate window functions

Date _started	Department	First _name	Salary	Dept_salary _budget	
2012-01-01	Finance	Martha	48500	48500	01.
2014-05-01	Finance	Joshua	35760	84260	02.
2015-06-01	Finance	Lily	35760	120020	03.
2017-06-15	Data_analytics	Emily	37800	37800	04.
2019-05-01	Data_analytics	Alex	36200	74000	
2022-03-01	Data_analytics	Maryam	46200	120200	
2016-01-01	Marketing	David	31000	31000	
2020-08-01	Marketing	Gabriel	30500	61500	
2022-07-15	Marketing	Sophia	36900	98400	

Employees' salaries are added together in each department in a running total.

01. The total salary budget in 2012 was R48500, which was only for Martha.

02. In 2014 we hired Joshua, and the total salaries increased to R84260.

03. Finally, in 2015 we hired Lily, so now we need to spend R120020 on salaries per month in Finance.

04. Note that the sum resets for each department and calculates a new running total.

Window functions in SQL

Ranking window functions

> Ranking window functions assign a **rank** or **row number** to each row within a specified window or subset of rows. Ranking window functions typically need an `ORDER BY` clause in order to work as intended.

`RANK()`

Ranks rows in each specified partition.

Duplicate rows are assigned the same rank, and the next **rank is skipped**, taking duplicates into account.

e.g., 1, 2, 2, 4.

`DENSE_RANK()`

Ranks rows in each specified partition.

Duplicate rows are assigned the same rank, but the ranks are **sequential regardless** of duplicates.

e.g., 1, 2, 2, 3.

`ROW_NUMBER()`

Assigns a **unique number** to each row within each partition, even if values in the partitioned column(s) are duplicated.

e.g., The sequence is 1, 2, 3, ..., regardless of duplicates.

`NTILE()`

Divides sorted partitions into `n` number of equal **groups**. Each row in a partition is assigned a group number 1, 2, 3...

e.g. if you have 100 rows of data, `NTILE(4)` will divide the data into `4 x 25` row groups.

The RANK() function

> The RANK() function assigns a rank to each row based on the order specified within the window. **Rows with the same values receive the same rank**, and **the next rank is skipped**.

Query

```
SELECT
    First_name,
    Province,
    RANK() OVER (
        ORDER BY Province) AS Rank_assign
FROM
    Employee
ORDER BY
    Province;
```

Suppose we want to rank each row by Province.

Note that we didn't use PARTITION BY here, so we will rank **all** rows together.

The RANK() function

First_name	Province	Rank_assign
Martha	Free_State	1
Maryam	Free_State	1
Emily	Gauteng	3
Lily	Gauteng	3
Sophia	Gauteng	3
Alex	Western_Cape	6
David	Western_Cape	6
Gabriel	Western_Cape	6
Joshua	Western_Cape	6

5 rows

01. Rows with the same values (`Free_State`) receive the same rank (rank 1).

02. Rank 2 is skipped because `Maryam` fell under rank 1.

03. Ranks 4 and 5 are skipped because there are **5 rows** above.

The DENSE_RANK() function

> The `DENSE_RANK()` function operates similarly to the `RANK()` function except it **does not skip** any ranks even if rows have the same values.

Query

```
SELECT
   First_name,
   Province,
   DENSE_RANK() OVER (
      ORDER BY Province) AS Rank_assign
FROM
   Employee
ORDER BY
   Province;
```

Suppose we want to rank each row by `Province,` but keep a sequential list, not skipping ranks.

Note that we didn't use `PARTITION BY` here, so we will rank **all** rows together.

Window functions in SQL

The DENSE_RANK() function

First_name	Province	Rank_assign	
Martha	Free_State	1	01.
Maryam	Free_State	1	
Emily	Gauteng	2	02.
Lily	Gauteng	2	
Sophia	Gauteng	2	
Alex	Western_Cape	3	03.
David	Western_Cape	3	
Gabriel	Western_Cape	3	
Joshua	Western_Cape	3	

01. Rows with the same values (`Free_State`) receive the same rank (rank 1).

02. Ranks are sequential here, even though there are duplicate values.

03. Since there are three unique values for `Province`, our rank goes up to 3.

ℹ Use `DENSE_RANK()` to avoid rank gaps and potential confusion. Use `RANK()` when maintaining relative differences between ranks is essential.

The ROW_NUMBER() function

> The ROW_NUMBER() function assigns a **unique sequential number** to each row within a partition, regardless of the column values. It makes sure that **no two rows can have the same row number** within a division.

Query

```
SELECT
   First_name,
   Province,
   ROW_NUMBER() OVER (
      PARTITION BY Province
      ORDER BY First_name) AS Row_assign
FROM
   Employee
ORDER BY
   Province;
```

Suppose we want to assign a unique number to each employee in a province based on their First_name.

Note that we add PARTITION BY Province here to reset the function for each partition.

The ROW_NUMBER() function

First_name	Province	Row_assign	
Martha	Free_State	1	01.
Maryam	Free_State	2	
Emily	Gauteng	1	02.
Lily	Gauteng	2	
Sophia	Gauteng	3	
Alex	Western_Cape	1	03.
David	Western_Cape	2	
Gabriel	Western_Cape	3	
Joshua	Western_Cape	4	

01. Sequential row assignment in the `Free_State` partition.

02. The row sequence resets in the `Gauteng` partition.

03. Sequential row assignment in the `Western_Cape` partition.

Each employee has a **unique** row **number** in their respective provinces.

Window functions in SQL

The NTILE() function

> `NTILE()` **divides** sorted partitions into n-number of **equal groups**. **Each row** in a partition is **assigned** a group **number**.

Query

```
SELECT
   First_name,
   Province,
   NTILE(2) OVER (
      PARTITION BY Province
      ORDER BY First_name) AS Group_number
FROM
   Employee
ORDER BY
   Province;
```

Suppose we want to divide employees from each province into two groups.

Since we are using `PARTITION BY` here, the data are split into partitions, and then `NTILE(2)` divides each `Province` into 2 groups and assigns a group number to each employee.

Window functions in SQL

The NTILE() function

First_name	Province	Group_number
Martha	Free_State	1
Maryam	Free_State	2
Emily	Gauteng	1
Lily	Gauteng	1
Sophia	Gauteng	2
Alex	Western_Cape	1
David	Western_Cape	1
Gabriel	Western_Cape	2
Joshua	Western_Cape	2

01. Employees are assigned a group number per department.

02. If rows **cannot split** equally, they are always **assigned** to the **first group**.

03. Here the partition can be equally subdivided.

The LAG() function

> The `LAG(column, n)` function allows access of a value within a column from the **previous** n^{th}-row **relative** to the **current row**.

Query

```
SELECT
   Department,
   First_name,
   Salary,
   LAG(Salary,1) OVER (
      ORDER BY Date_started) AS Previous_salary
FROM
   Employee
ORDER BY
   Department;
```

Suppose we want to retrieve the previous salaries according to the employee's date of hire.

The `LAG(Salary, 1)` function is applied to the `Salary` column and ordered by `Date_started`. It returns the salary from the previous row, since `n = 1,` as the column `Previous_salary`.

Window functions in SQL

The LAG() function

Department	First_name	Salary	Previous_salary
Data_analytics	Maryam	46200	NULL
Data_analytics	Alex	36200	46200
Data_analytics	Emily	37800	36200
Finance	Joshua	35760	37800
Finance	Lily	35760	35760
Finance	Martha	48500	35760
Marketing	Sophia	36900	48500
Marketing	David	31000	36900
Marketing	Gabriel	30500	31000

The value on the first row of a partition will always be `NULL` since its previous value does not exist.

Values from the `Previous_salary` column were the `Salary` values from the **previous row** `(n = 1)`.

We can now use these values to calculate things like rates of change, difference, etc.

Window functions in SQL

The LEAD() function

> The LEAD(column, n) function allows access of a value within a column from the **following** n^{th}-row **relative** to the **current row**. It is the counterpart of the LAG() function.

Query

```
SELECT
    Department,
    First_name,
    Salary,
    LEAD(Salary,1) OVER (
        PARTITION BY Department
        ORDER BY Date_started) AS Next_salary
FROM
    Employee
ORDER BY
    Department;
```

Suppose we want to retrieve the next employee's name according to the employee's salary.

The LEAD(Salary, 1) function is applied to the Salary column and ordered by Date_started. It returns the salary from the previous row, since n = 1, as the column Next_salary.

The LEAD() function

Department	First_name	Salary	Next_salary
Data_analytics	Maryam	46200	36200
Data_analytics	Alex	36200	37800
Data_analytics	Emily	37800	NULL
Finance	Joshua	35760	35760
Finance	Lily	35760	48500
Finance	Martha	48500	NULL
Marketing	Sophia	36900	31000
Marketing	David	31000	30500
Marketing	Gabriel	30500	NULL

01. Values from the Next_salary column are the `Salary` values from the **next row** (n = 1), which is the reverse of `LAG()`.

02. Since we used `PARTITION BY`, each department is a separate partition. The function applies to each separately.

03. The last row of a partition will always be `NULL` since the function cannot access data from another partition.

The FIRST_VALUE() function

> The FIRST_VALUE() function allows the retrieval of the value of a column from the **first row** within a partition.

Suppose we want to retrieve the first employee the company hired.

Query

```
SELECT
   Start_date,
   Department,
   First_name,
   FIRST_VALUE(First_name) OVER (
      ORDER BY Start_date
      PARTITION BY Department) AS First_in_dept
FROM
   Employee
ORDER BY
   Department;
```

The FIRST_VALUE() function is applied to the First_name column and ordered by Start_date. It returns the First_name from the first row as First_in_dept.

Window functions in SQL

The FIRST_VALUE() function

Start_date	Department	First _name	First _in_dept
2017-06-15	Data_analytics	Emily	Emily
2019-05-01	Data_analytics	Alex	Emily
2022-03-01	Data_analytics	Maryam	Emily
2012-01-01	Finance	Martha	Martha
2014-05-01	Finance	Joshua	Martha
2015-06-01	Finance	Lily	Martha
2016-01-01	Marketing	David	David
2020-08-01	Marketing	Gabriel	David
2022-07-15	Marketing	Sophia	David

01. Only the first value, or the first hired employee, `Emily`, will be the output in the Data_analytics partition.

02. Since we used `PARTITION BY Department`, the first employee from **each department** is returned.

The LAST_VALUE() function

> The `LAST_VALUE()` function allows the retrieval of the value of a column from the **last row** within a window frame.

Query

```
SELECT
   Start_date,
   Department,
   First_name,
   LAST_VALUE(First_name) OVER (
      ORDER BY Start_date) AS Last_employee
FROM
   Employee
ORDER BY
   Department;
```

Suppose we want to retrieve the **last employee** the company hired.

The `LAST_VALUE()` function is applied to the `First_name` column and ordered by `Start_date`. It returns the `First_name` from the first row as Last_employee.

Window functions in SQL

The LAST_VALUE() function

Start_date	Department	First_name	Last_employee
2017-06-15	Data_analytics	Emily	Emily
2019-05-01	Data_analytics	Alex	Alex
2022-03-01	Data_analytics	Maryam	Maryam
2012-01-01	Finance	Martha	Martha
2014-05-01	Finance	Joshua	Joshua
2015-06-01	Finance	Lily	Lily
2016-01-01	Marketing	David	David
2020-08-01	Marketing	Gabriel	Gabriel
2022-07-15	Marketing	Sophia	Sophia

Sophia will appear at the end of the row as she is the last employee.