EXPLORE AI ACADEMY

SQL window functions

Window functions and syntax

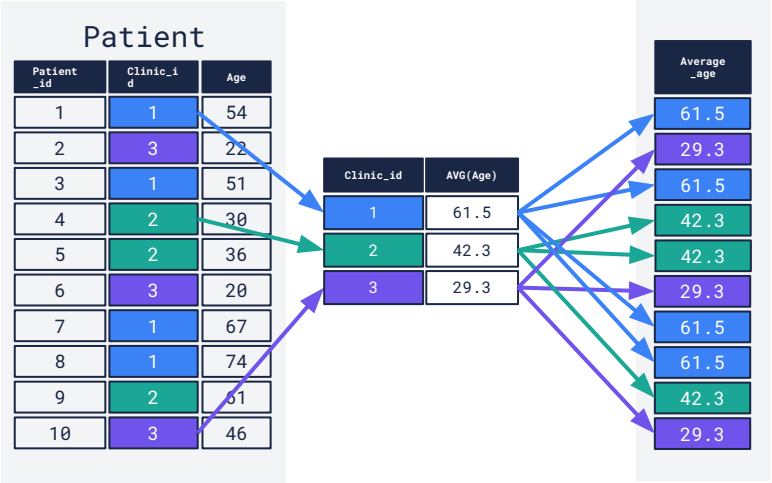
Window functions and syntax

What are window functions?

SQL aggregate functions are powerful tools **that summarise** data across partitions. They provide valuable insights but often **restrict** detailed **analysis** due to their summary format.

SQL **window functions** address this limitation, conducting operations across related row sets or windows. They **allow** for more **nuanced** tasks, like calculating running totals or finding maximum group values, extending the capabilities of standard aggregation functions.

In this example, we calculate the average age of patients per clinic, but instead of viewing the aggregated results, we use a window function to create a new column with the average age now in each row, depending on which clinic the patient visited.



Window function syntax

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

Window function syntax

Syntax of window functions

```
SELECT
   Column1,
   …,
   WIN_FUNCTION(arg) OVER(
     PARTITION BY ColumnX
     ORDER BY ColumnY)
FROM
   Db_name.Table_name;
```

Window functions use data from specified **row sets** or **windows**. `argument` specifies columns or parameters passed to the function.

Window function examples

Aggregate

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

Ranking

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

Analytical

- `LAG()`
- `LEAD()`
- `AVG()`
- `SUM()`

Window function syntax

Syntax of window functions

```
SELECT
    Column1,
    …,
    WIN_FUNCTION(arg) OVER(
      PARTITION BY ColumnX
      ORDER BY ColumnY)
FROM
    Db_name.Table_name;
```

This keyword indicates that you are using a window function.

(OPTIONAL)
We use `PARTITION BY` when we want to divide our result set into **partitions/windows** or **subsets**, and **apply** a **function** to each **window**, based on the values in `ColumnX`.
This is similar to how `GROUP BY` works with aggregate functions, except `PARTITION BY` keeps the individual row data within each partition.

Window function syntax

Syntax of window functions

```
SELECT
   Column1,
   …,
   WIN_FUNCTION(arg) OVER(
     PARTITION BY ColumnX
     ORDER BY ColumnY)
FROM
   Db_name.Table_name;
```

(OPTIONAL)
We use `ORDER BY` within a window function to **sort the rows** within each partition by the values in `ColumnY`.

This is **important** for **ranking functions** where the order of rows affects the result. Using `ORDER BY` with **aggregate functions** we can calculate running metrics like running totals (sum of all values in a partition, up to that row) or running averages.

Example: Using PARTITION BY

> Suppose we want to compare each employee's salary with the average salary of the corresponding department.

Query

```
SELECT Department,
       First_name,
       Salary,
       AVG(Salary) OVER (PARTITION BY Department)
       AS Average_salary
FROM
      Employee;
```

This aggregate window function calculates the **average salary** from the `Salary` column within each partition, defined by the `Department` column.

In other words, the average will be calculated based on the whole set of rows **within** each **partition**.

Window function syntax

Example: Using PARTITION BY

Department	First_name	Salary	Average_salary
Data_analytics	Emily	37800	40067
Data_analytics	Alex	36200	40067
Data_analytics	Maryam	46200	40067
Finance	Joshua	35760	40007
Finance	Lily	35760	40007
Finance	Martha	48500	40007
Marketing	David	31000	32800
Marketing	Gabriel	30500	32800
Marketing	Sophia	36900	32800

Manual calculation:
`(37800 + 36200 + 46200)/3 =` **R 40067**

Manual calculation:
`(35760 + 35760 + 48500)/3 =` **R 40007**

Manual calculation:
`(31000 + 30500 + 36900)/3 =` **R 32800**

This result set has been arranged to make the point clear. SQL will return a table with the rows in their original positions.

Example: ORDER BY

Suppose we want to **rank** employees according to **salary** across the **whole** organisation.

Query

```
SELECT
    Department,
    First_name,
    Salary,
    RANK() OVER (
        ORDER BY Salary DESC) AS Rank_
FROM
    Employee;
```

`RANK()` assigns a unique rank to each row within a partition, ordered by the specified column, such as `Salary`.

When `RANK()` assigns a rank to a row, **duplicate values are "tied"** and so they receive the **same rank** and then the next rank is skipped.

Since there are **no partitions**, `ORDER BY` will just rank all rows together.

Window function syntax

Example: ORDER BY

Department	First_name	Salary	Rank_
Finance	Martha	48500	1
Data_analytics	Maryam	46200	2
Data_analytics	Emily	37800	3
Marketing	Sophia	36900	4
Data_analytics	Alex	36200	5
Finance	Lily	35760	6
Finance	Joshua	35760	6
Marketing	David	31000	8
Marketing	Gabriel	30500	9

Since there are no partitions, `RANK()` ranks all of the rows together.

Note that Lily and Joshua have the same salaries, so they share rank 6, and David is 8th.

Example: ORDER BY

> Suppose we want to rank employees in **each department** from lowest to highest salary. We add `PARTITION BY Department`, and then `ORDER BY Salary`.

Query

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

`RANK()` arranges rows **in each partition**, specified by the `PARTITION BY` column (`Department`), based on the column we specify in the `ORDER BY` column (Salary).

By adding `PARTITION BY`, we will rank employees in **each department separately**.

Window function syntax

Example: ORDER BY

Department	First_name	Salary	Rank
Data_analytics	Maryam	46200	1
Data_analytics	Alex	36200	2
Data_analytics	Emily	37800	3
Finance	Joshua	35760	1
Finance	Lily	35760	1
Finance	Martha	48500	3
Marketing	Sophia	36900	1
Marketing	David	31000	2
Marketing	Gabriel	30500	3

Each employee is ranked in their own department.

Joshua and Lily have the same salaries, so they are both "tied" for rank 1. `RANK()` skips the next rank, and assigns rank 3 to Marta.

Manual calculation:
`(31000 + 30500 + 36900)/3 = R 32800`

This result set has been arranged to make the point clear. SQL will return a table with the rows in their original positions.

Window function syntax

Aggregate window functions with ORDER BY

> Suppose we want to compare the average salaries for each department over time. We `PARTITION BY Department`, and then `ORDER BY Date_started`.

Query

```
SELECT
    Date_started,
    Department,
    First_name,
    Salary,
    AVG(Salary) OVER (PARTITION BY Department
    ORDER BY Date_started) AS Average_salary
FROM
    Employee;
```

Using `PARTITION BY` and `ORDER BY` with an **aggregate function** orders each partition and we get "running" metrics in the result set, like running totals or running averages.

By using `ORDER BY` with an **aggregate function**, we calculate the **running average of** `Salary` for each department that changes as we hire more employees.

Window function syntax

Aggregate window functions with ORDER BY

Date _started	Department	First_ name	Salary	Average _salary
2012-01-01	Finance	Martha	48500	48500
2014-05-01	Finance	Joshua	29500	39000
2015-06-01	Finance	Lily	35760	37920
2016-01-01	Marketing	David	31000	31000
2020-08-01	Marketing	Gabriel	30500	30750
2022-07-15	Marketing	Sophia	36900	32700
2017-06-15	Data_analytics	Emily	37800	37800
2019-05-01	Data_analytics	Alex	36200	37000
2022-03-01	Data_analytics	Maryam	46200	40033

01. **Average salary = salary for Martha**

02. **Average salary between Martha and Joshua**
`(48500 + 29500)/2 = R 39000`

03. **Average salary among Martha, Joshua, and Lily**
`(48500 + 39000 + 35760)/3 = R 37920`

04. `Average_salary` resets to **R 31000** in the `Marketing` partition.

This result set has been arranged to make the point clear. SQL will return a table with the rows in their original positions.