Chapter 6 : Window Functions

Aviral Janveja

1 Introduction

Window functions perform operations over groups of rows, also called windows. This sounds similar to aggregate functions and the group-by clause. However, while group-by collapses each group into a single resultant row, window functions simply add the result to every row. Hence, we keep the individual row data as it is, and get the aggregated data alongside. The following query uses window functions to return the department-wise average salaries :

```
SELECT
    employee_no
    department
    salary
    AVG(salary) OVER(PARTITION BY department)
        AS dept_avg
FROM
    employees
```

We can do the same for other aggregate functions as well. But for now, let us break down the syntax, specific to window functions.

2 OVER

The over clause creates the window. When it is empty, the window will include all rows. For example :

```
AVG(salary) OVER()
```

The over clause tells SQL that we are running this as a window function. Leaving it empty creates one massive window that includes all the rows in our table. Hence, the above query will calculate the overall average salary for the given company and append it to each row.

3 PARTITION BY

Inside the over clause, you can use partition-by to form smaller windows. For example :

```
AVG(salary) OVER(PARTITION BY department)
```

Instead of the overall average, the above query now gives us the department average.

4 ORDER BY

Further, you can use the order-by clause inside the over clause to reorder rows within each window. For example :

```
RANK() OVER(PARTITION BY department
        ORDER BY salary DESC) AS dept_rank
```

The above query ranks employees according to their salaries, department wise, from the highest to the lowest. The above result cannot be achieved via the group-by clause. Next, let us look at some more window functions that can only be used with the over clause.

5 RANK and DENSE_RANK

The RANK( ) function ranks the rows based on values in a given column. For example, by employee salaries as shown in the above example. DENSE_RANK( ) does the same thing but without skipping ranks.

For example, if two rows have an identical rank 4, the following row will be given the rank 6 by the rank function. Whereas, dense rank will rank it as 5 without skipping any ranks.

6 ROW_NUMBER

The ROW_NUMBER( ) function displays the number of a given row as per the order-by part of the window statement. The partition-by clause will make the count restart from 1 with each partition.

7 NTILE

You can use the NTILE function to identify what bucket or subdivision a given row falls into. The syntax is as follow :

```
NTILE(4) OVER() AS quartile
NTILE(5) OVER() AS quintile
NTILE(100) over() AS percentile
```

Here, the order-by clause determines which column will be used to calculate the quartile or percentile for a given row.

8 LAG and LEAD

It can often be useful to compare the current row to its preceding or following rows. LAG( ) pulls from previous rows and LEAD( ) pulls from following rows. All you need to do is specify which column to pull from and how many rows away you would like to do the pull. For example :

```
LAG(salary, 1) OVER() AS lag
LEAD(salary, 1) OVER() AS lead
```

This is especially useful if you want to calculate differences between rows as follows :

```
SELECT
    salary,
    salary - LAG(salary, 1) OVER() AS difference

FROM
    employees
```

9 Window Alias

If you are planning to write several window functions in a query, using the same window, you can create an alias instead of typing the whole thing again. For example :

```
SELECT
    employee_no
    department
    salary
    AVG(salary) OVER common_window AS dept_avg
    MAX(salary) OVER common_window AS dept_max
    MIN(salary) OVER common_window AS dept_min

FROM
    employees

WINDOW
    common_window AS (PARTITION BY department
    ORDER BY salary DESC)
```

The window clause if included, should always come after the where clause.