2018-08-09T09:57:55+01:00 Ignacio L. Bisso

SQL Window Function Example With Explanations

Interested in how window functions work? Scroll down to see our SQL window function example with definitive explanations!

SQL window functions are a bit different; they compute their result based on a **set of rows** rather than on a single row. In fact, the "window" in "window function" refers to that set of rows.

Window functions are **similar to aggregate functions**, but there is one important difference. When we use aggregate functions with the `GROUP BY` clause, we "lose" the individual rows. We can't mix attributes from an individual row with the results of an aggregate function; the function is performed on the rows as an entire group. This is not the case when we use SQL window functions: we can generate a result set with some attributes of an individual row together with the results of the window function. This is good for **new SQL developers** to keep in mind. So let's examine a simple *SQL window function example* in action.

Want to learn SQL window functions? Check out our interactive Window Functions course!

Want to learn about window functions? Click here for a great interactive experience!

SQL Window Function Example

Window functions can be called in the `SELECT` statement or in the `ORDER BY` clause. However, they can never be called in the `WHERE` clause. You'll notice that all the examples in this article call the **window function** in the `SELECT` column list.

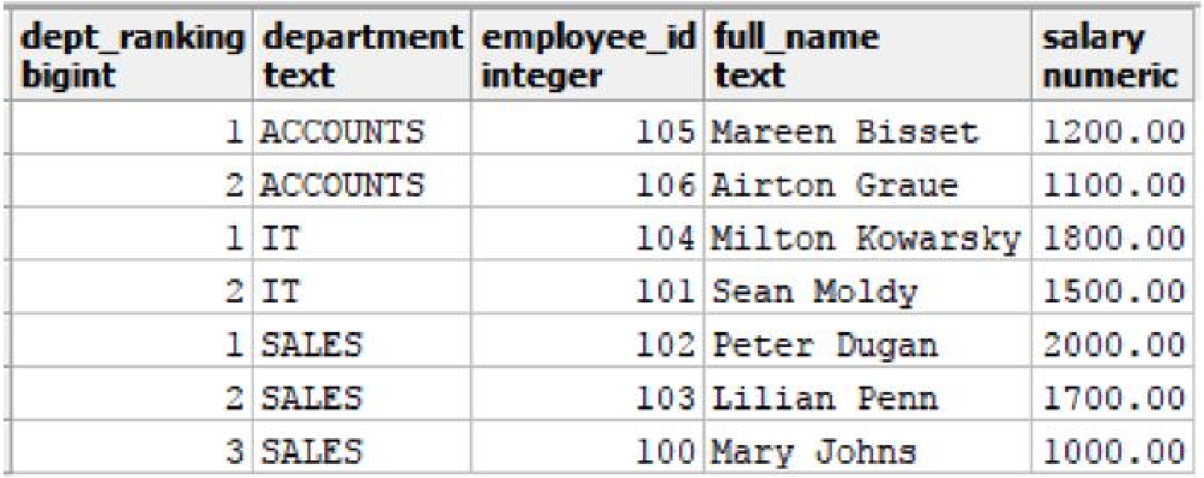
Let’s go to the first SQL window function example. We will use the “**`Employee`**” table:

employee_id	full_name	department	salary
100	Mary Johns	SALES	1000.00
101	Sean Moldy	IT	1500.00
102	Peter Dugan	SALES	2000.00
103	Lilian Penn	SALES	1700.00
104	Milton Kowarsky	IT	1800.00
105	Mareen Bisset	ACCOUNTS	1200.00
106	Airton Graue	ACCOUNTS	1100.00

We will begin with RANK, which is one of the simplest *SQL window functions* example. It returns the position of any row inside the partition. Let’s use it to rank salaries within departments:

```
SELECT
RANK() OVER (PARTITION BY department ORDER BY salary DESC)
AS dept_ranking,
department,
employee_id,
full_name,
salary
FROM employee;
```

We can see the results below:



dept_ranking bigint	department text	employee_id integer	full_name text	salary numeric
1	ACCOUNTS	105	Mareen Bisset	1200.00
2	ACCOUNTS	106	Airton Graue	1100.00
1	IT	104	Milton Kowarsky	1800.00
2	IT	101	Sean Moldy	1500.00
1	SALES	102	Peter Dugan	2000.00
2	SALES	103	Lilian Penn	1700.00
3	SALES	100	Mary Johns	1000.00

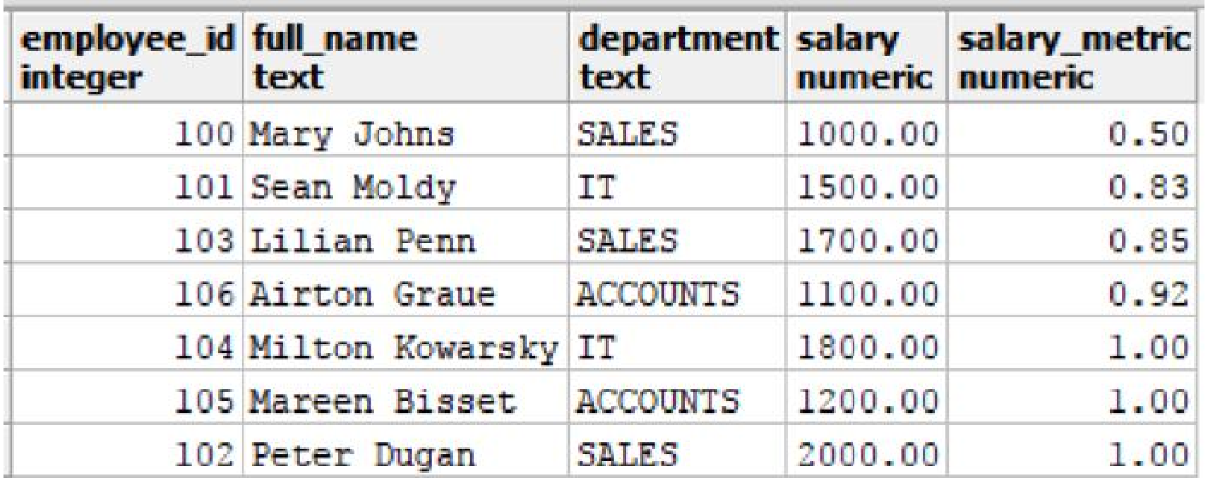
What if we want to have the same report but with all the top-ranking employees first, then all second-ranking employees, and so on? Well, we'll give you this challenge to figure out on your own. Share your ideas in the comments section!

Proceeding with our SQL window function example, let's find out where each employee's salary ranks in relation to the top salary of their department. This calls for a math expression, like:

employee_salary / max_salary_in_depth

The next query will show all employees ordered by the above metric; the employees with the lowest salary (relative to their highest departmental salary) will be listed first:

```
SELECT
employee_id,
full_name,
department,
salary,
salary / MAX(salary) OVER (PARTITION BY department ORDER BY
salary DESC)
AS salary_metric
FROM employee
ORDER BY 5;
```



employee_id integer	full_name text	department text	salary numeric	salary_metric numeric
100	Mary Johns	SALES	1000.00	0.50
101	Sean Moldy	IT	1500.00	0.83
103	Lilian Penn	SALES	1700.00	0.85
106	Airton Graue	ACCOUNTS	1100.00	0.92
104	Milton Kowarsky	IT	1800.00	1.00
105	Mareen Bisset	ACCOUNTS	1200.00	1.00
102	Peter Dugan	SALES	2000.00	1.00

Interested in learning SQL window functions? Try out our interactive Window Functions course!

Learn window functions through practice. Our Window Functions course has 218 interactive exercises. Try it out!

Another SQL Window Function Example

Let's switch from an employee-salary database to the following train schedule database:

Train_id	Station	Time
110	San Francisco	10:00:00
110	Redwood City	10:54:00
110	Palo Alto	11:02:00
110	San Jose	12:35:00
120	San Francisco	11:00:00
120	Redwood City	Non Stop
120	Palo Alto	12:49:00
120	San Jose	13:30:00

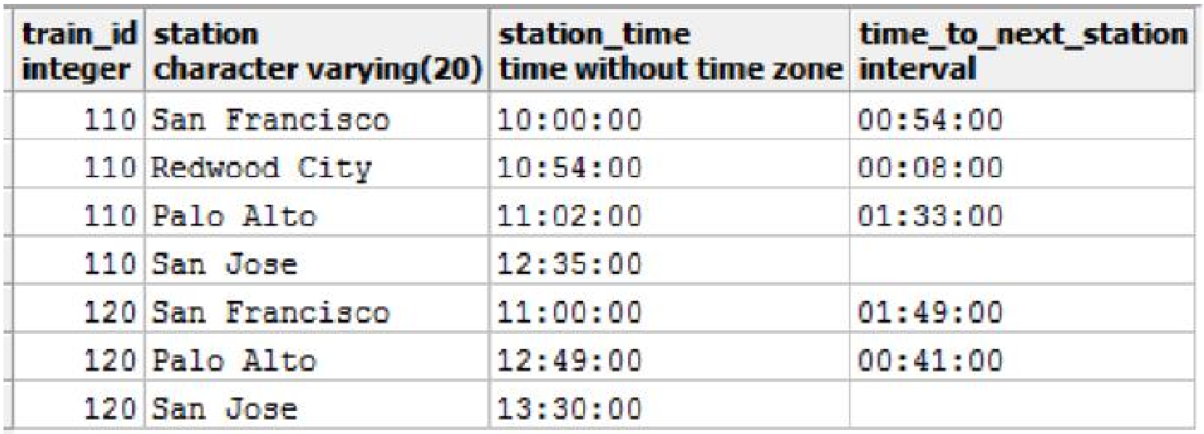
Suppose we want to add a new column called "time to next station". To obtain this value, we subtract the station times for pairs of contiguous stations. We can calculate this value without using a SQL window function, but that can be very complicated. It's simpler to do it using the LEAD *window function*. This function compares values from one row with the next row to come up with a result. In this case, it compares the values in the "time" column for a station with the station immediately after it.

So, here we have another SQL window function example, this time for the train schedule:

```
SELECT
train_id,
station,
time as "station_time",
lead(time) OVER (PARTITION BY train_id ORDER BY time) - time
AS time_to_next_station
FROM train_schedule;
```

Note that we calculate the `LEAD` window function by using an expression involving an individual column **and** a *window function*; this is not possible with aggregate functions.

Here are the results of that query:

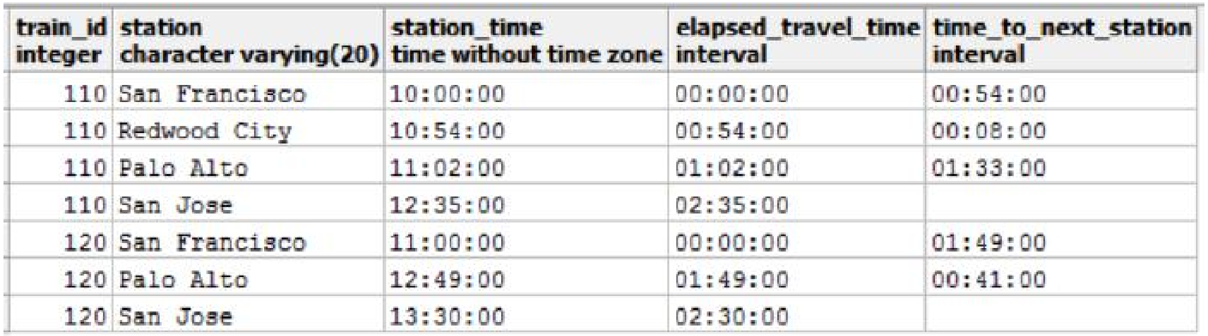


train_id integer	station character varying(20)	station_time time without time zone	time_to_next_station interval
110	San Francisco	10:00:00	00:54:00
110	Redwood City	10:54:00	00:08:00
110	Palo Alto	11:02:00	01:33:00
110	San Jose	12:35:00	
120	San Francisco	11:00:00	01:49:00
120	Palo Alto	12:49:00	00:41:00
120	San Jose	13:30:00	

In the next example, we will add a new column that shows how much time has elapsed from the train's first stop to the current station. We will call it "elapsed travel time". The MIN window function will obtain the trip's start time and we will subtract the current station time. Here's the next SQL window function example

SELECT
train_id,
station,
time as "station_time",
time - min(time) OVER (PARTITION BY train_id ORDER BY time)
AS elapsed_travel_time,
lead(time) OVER (PARTITION BY train_id ORDER BY time) - time
AS time_to_next_station
FROM train_schedule;

Notice the new column in the result table:



train_id integer	station character varying(20)	station_time time without time zone	elapsed_travel_time interval	time_to_next_station interval
110	San Francisco	10:00:00	00:00:00	00:54:00
110	Redwood City	10:54:00	00:54:00	00:08:00
110	Palo Alto	11:02:00	01:02:00	01:33:00
110	San Jose	12:35:00	02:35:00	
120	San Francisco	11:00:00	00:00:00	01:49:00
120	Palo Alto	12:49:00	01:49:00	00:41:00
120	San Jose	13:30:00	02:30:00	