Window Functions

A *window function* performs a calculation across a set of table rows that are somehow related to the current row.

This is comparable to the type of calculation that can be done with an aggregate function.

But unlike regular aggregate functions, use of a window function does not cause rows to become grouped into a single output row — the rows retain their separate identities.

Behind the scenes, the window function is able to access more than just the current row of the query result.

Before starting window functions let's have a quick overview of aggregate functions and their uses in SQL.

SQL Aggregate functions

What Are SQL Aggregate Functions?

Aggregate functions operate on a set of values to return **a single scalar value**. These are SQL aggregate functions:

- AVG() returns the average of the specified values.
- SUM() calculates the sum of all values in the set.
- MAX() and MIN() return the maximum and minimum value, respectively.
- COUNT() returns the total number of values in the set.

Aggregate functions summarize data from multiple rows into a single result row.

For example, the following SUM ( ) function returns the total sales of all employees in the recorded years:

```
SELECT
    SUM(sale)
FROM
    sample_db.sales;
```

```
ts 1
T SUM(sale) FROM sample_db
```

123 sum
1,500

By using the GROUP BY clause, we can calculate an aggregate value for several groups in one query. For example, we may want to calculate the total sales by fiscal years:

```
SELECT
    fiscal_year,
    SUM(sale)
FROM
    sample_db.sales
GROUP BY
    fiscal_year;
```

```
s 1
CT fiscal_year, SUM(sale) FROM sample
```

123 fiscal_year	123 sum
2,017	400
2,016	450
2,018	650

We can see both the above examples.

Note- The aggregate functions reduces the number of rows returned by the query.

Like the aggregate functions with the GROUP BY clause, window functions also operate on a subset of rows but they do not reduce the number of rows returned by the query.

For example, the following query returns the sales for each employee, along with total sales of the employees by fiscal year:

```sql
SELECT
    fiscal_year,
    sales_employee,
    sale,
    SUM(sale) OVER (PARTITION BY fiscal_year) total_sales
FROM
    sample_db.sales;
```

1

CT fiscal_year, sales_employee, sale, SU | Enter a SQL expression to filter results (u

123 fiscal_year	ABC sales_employee	123 sale	123 total_sales
2,016	Alice	150	450
2,016	Bob	100	450
2,016	John	200	450
2,017	Alice	100	400
2,017	Bob	150	400
2,017	John	150	400
2,018	Bob	200	650
2,018	John	250	650
2,018	Alice	200	650

Note that window functions are performed on the result set after all JOIN, WHERE, GROUP BY, and HAVING clauses and before the ORDER BY, LIMIT and SELECT DISTINCT.

Window functions Syntax

The general syntax of calling a window function is as follows:

```
window_function_name(expression) OVER (
    [partition_defintion]
    [order_definition]
    [frame_definition]
)
```

In this syntax:

- First, specify the window function name followed by an expression.

- Second, specify the OVER clause which has three possible elements: partition definition, order definition, and frame definition.

The opening and closing parentheses after the OVER clause are mandatory, even with no expression, for example:

```
window_function_name(expression) OVER()
```

Partition **clause**

The partition clause breaks up the rows into chunks or partitions. Two partitions are separated by a partition boundary.

It is similar to group by.

e.g.

In below query we are partitioning the whole table based on fiscal_year column and finding year wise aggregate sum.

```
select sales_employee,fiscal_year,sale ,
SUM(sale)
over (partition by fiscal_year) as total_sales
from sample_db.sales;
```

sales 1

select sales_employee,fiscal_year,sale , SUM(sale) c | Enter a SQL expression to filter resu

	sales_employee	fiscal_year	sale	total_sales
1	Alice	2,016	150	450
2	Bob	2,016	100	450
3	John	2,016	200	450
4	Alice	2,017	100	400
5	Bob	2,017	150	400
6	John	2,017	150	400
7	Bob	2,018	200	650
8	John	2,018	250	650
9	Alice	2,018	200	650

Order by **clause**

The ORDER BY clause specifies how the rows are ordered within a partition.

```sql
select sales_employee,fiscal_year,sale,
row_number() over( order by sale desc) as row_num
from sample_db.sales;
```

sales 1

select sales_employee,fiscal_year,sale, row_numbe | Enter a SQL expression to filter resu

	sales_employee	fiscal_year	sale	row_num
1	John	2,018	250	1
2	Bob	2,018	200	2
3	Alice	2,018	200	3
4	John	2,016	200	4
5	Alice	2,016	150	5
6	John	2,017	150	6
7	Bob	2,017	150	7
8	Alice	2,017	100	8
9	Bob	2,016	100	9

In above query and its result we can clearly see the ordering due to the order by clause.

Note- PARTITION BY clause and the ORDER BY clause is also supported by all the window functions.

Some Important window functions

Aggregate Window Function :

Various aggregate functions such as SUM(), COUNT(), AVERAGE(), MAX(), MIN() applied over a particular window (set of rows) are called aggregate window functions. Consider the below sales table:

sales_employee	fiscal_year	sale
Bob	2,016	100
Bob	2,017	150
Bob	2,018	200
Alice	2,016	150
Alice	2,017	100
Alice	2,018	200
John	2,016	200
John	2,017	150
John	2,018	250

```
select sales_employee,fiscal_year,sale ,
SUM(sale)
over (partition by fiscal_year order by sale desc) as total_sales
from sample_db.sales;
```

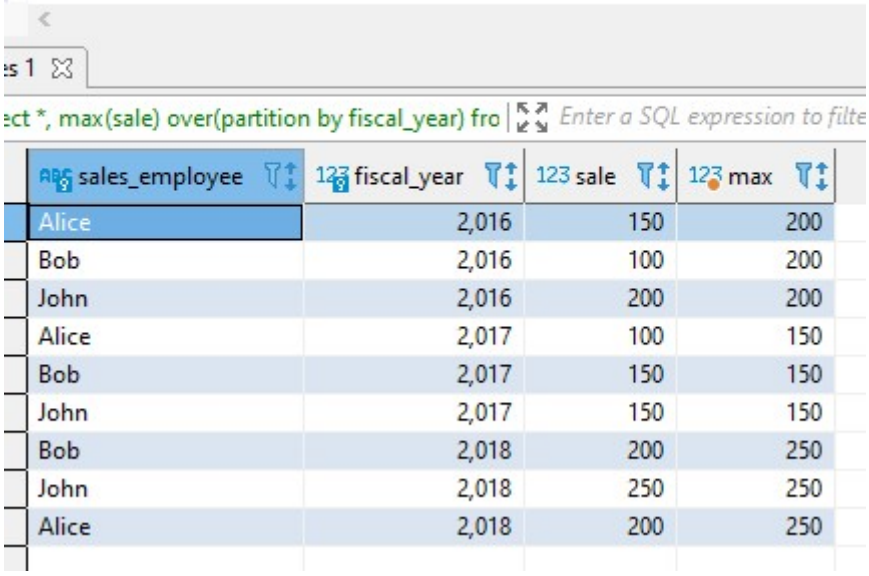
les 1

lect sales_employee,fiscal_year,sale , SUM(sale) c | Enter a SQL expression to filter results (u

sales_employee	fiscal_year	sale	total_sales
John	2,016	200	200
Alice	2,016	150	350
Bob	2,016	100	450
Bob	2,017	150	300
John	2,017	150	300
Alice	2,017	100	400
John	2,018	250	250
Alice	2,018	200	650
Bob	2,018	200	650

Above SQL query we are using SUM() as window function to generate total sales fiscal year wise.

```
select *,
max(sale) over(partition by fiscal_year)
from sample_db.sales;
```



ect *, max(sale) over(partition by fiscal_year) fro | Enter a SQL expression to filte

sales_employee	fiscal_year	sale	max
Alice	2,016	150	200
Bob	2,016	100	200
John	2,016	200	200
Alice	2,017	100	150
Bob	2,017	150	150
John	2,017	150	150
Bob	2,018	200	250
John	2,018	250	250
Alice	2,018	200	250

Similarly we can use other aggregate functions as window functions.

ROW_NUMBER

Assigns a sequential integer to every row within its partition.

There must be order by clause.

Partition by clause is optional.

In each partition row number starts from 1.

```
select sales_employee,fiscal_year,sale,
row_number() over (order by sale desc) as rownum
from sample_db.sales;
```

sales 1

select sales_employee,fiscal_year,sale, row_numbe | Enter a SQL expression to filter re

	sales_employee	fiscal_year	sale	rownum
1	John	2,018	250	1
2	Bob	2,018	200	2
3	Alice	2,018	200	3
4	John	2,016	200	4
5	Alice	2,016	150	5
6	John	2,017	150	6
7	Bob	2,017	150	7
8	Alice	2,017	100	8
9	Bob	2,016	100	9

row_number() with partition clause

```
select sales_employee,fiscal_year,sale,
 row_number() over (partition by fiscal_year order by sale desc) as rownum
 from sample_db.sales;
```

sales 1

select sales_employee,fiscal_year,sale, row_numbe | Enter a SQL expression to filter results (use Ctrl+Sp

	sales_employee	fiscal_year	sale	rownum
1	John	2,016	200	1
2	Alice	2,016	150	2
3	Bob	2,016	100	3
4	Bob	2,017	150	1
5	John	2,017	150	2
6	Alice	2,017	100	3
7	John	2,018	250	1
8	Alice	2,018	200	2
9	Bob	2,018	200	3

LEAD

Returns the value of the Nth row after the current row in a partition. It returns NULL if no subsequent row exists.

Suppose we have below job_role table:

	123 id	ABC job_category	ABC country	123 salary
1	1	Data Scientist	India	80,000
2	2	ML Engineer	US	120,000
3	3	Developer	India	70,000
4	4	Data Analyst	India	65,000
5	5	Data Analyst	India	60,000
6	6	Developer	US	110,000
7	7	Data Scientist	US	150,000
8	8	Data Analyst	US	100,000
9	9	Data Scientist	UK	140,000
10	10	Data Scientist	UK	160,000
11	11	Data Analyst	UK	150,000
12	12	Data Scientist	US	200,000
13	13	Data Scientist	US	300,000
14	14	Developer	UK	151,000
15	15	Data Analyst	UK	101,000
16	16	Developer	UK	99,000
17	17	Developer	India	50,000
18	18	ML Engineer	India	55,000
19	19	ML Engineer	US	125,000
20	20	Developer	India	40,000

Suppose we need to write a query to display if the salary of an employee is higher, lower or equal to the previous employee.

Below query prints the LEAD salary.

```
select *,
lead(salary) over( order by id) as next_emp_salary
from sample_db.job_role ;
```

job_role 1

select *, lead(salary) over(order by id) as next_emp Enter a SQL expression to filter results (use Ctrl+Sp

	id	job_category	country	salary	next_emp_salary
1	1	Data Scientist	India	80,000	120,000
2	2	ML Engineer	US	120,000	70,000
3	3	Developer	India	70,000	65,000
4	4	Data Analyst	India	65,000	60,000
5	5	Data Analyst	India	60,000	110,000
6	6	Developer	US	110,000	150,000
7	7	Data Scientist	US	150,000	100,000
8	8	Data Analyst	US	100,000	140,000
9	9	Data Scientist	UK	140,000	160,000
10	10	Data Scientist	UK	160,000	150,000
11	11	Data Analyst	UK	150,000	200,000
12	12	Data Scientist	US	200,000	300,000
13	13	Data Scientist	US	300,000	151,000
14	14	Developer	UK	151,000	101,000
15	15	Data Analyst	UK	101,000	99,000
16	16	Developer	UK	99,000	50,000
17	17	Developer	India	50,000	55,000
18	18	ML Engineer	India	55,000	125,000
19	19	ML Engineer	US	125,000	40,000
20	20	Developer	India	40,000	[NULL]

LAG

Returns the value of the Nth row before the current row in a partition. It returns NULL if no preceding row exists.

Suppose we need to write a query to display if the salary of an employee is higher, lower or equal to the previous employee.

We can easily print the lagging salary with the LAG () window function as shown below.

```
select *,
lag(salary) over( order by id) as pre_emp_salary
from sample_db.job_role jr ;
```

job_role 1

select *, lag(salary) over(order by id) as pre_emp_ | Enter a SQL expression to filter results (use Ctrl+S

	id	job_category	country	salary	pre_emp_salary
1	1	Data Scientist	India	80,000	[NULL]
2	2	ML Engineer	US	120,000	80,000
3	3	Developer	India	70,000	120,000
4	4	Data Analyst	India	65,000	70,000
5	5	Data Analyst	India	60,000	65,000
6	6	Developer	US	110,000	60,000
7	7	Data Scientist	US	150,000	110,000
8	8	Data Analyst	US	100,000	150,000
9	9	Data Scientist	UK	140,000	100,000
10	10	Data Scientist	UK	160,000	140,000
11	11	Data Analyst	UK	150,000	160,000
12	12	Data Scientist	US	200,000	150,000
13	13	Data Scientist	US	300,000	200,000
14	14	Developer	UK	151,000	300,000
15	15	Data Analyst	UK	101,000	151,000
16	16	Developer	UK	99,000	101,000
17	17	Developer	India	50,000	99,000
18	18	ML Engineer	India	55,000	50,000
19	19	ML Engineer	US	125,000	55,000
20	20	Developer	India	40,000	125,000

DENSE_RANK

Assigns a rank to every row within its partition based on the ORDER BY clause. It assigns the same rank to the rows with equal values. If two or more rows have the same rank, then there will be no gaps in the sequence of ranked values.

```sql
-- dense_rank()
select sales_employee,fiscal_year,sale,
 dense_rank () over(order by sale desc) as denserank
 from sample_db.sales;

-- rank()
```

sales 1

select sales_employee,fiscal_year,sale, dense_rank | Enter a SQL expression to filter results

	sales_employee	fiscal_year	sale	denserank
1	John	2,018	250	1
2	Bob	2,018	200	2
3	Alice	2,018	200	2
4	John	2,016	200	2
5	Alice	2,016	150	3
6	John	2,017	150	3
7	Bob	2,017	150	3
8	Alice	2,017	100	4
9	Bob	2,016	100	4

RANK

Similar to the DENSE_RANK() function except that there are gaps in the sequence of ranked values when two or more rows have the same rank.

```sql
-- rank()
select sales_employee,fiscal_year,sale,
 rank() over (order by sale desc ) as ranknum
 from sample_db.sales;
```

ales 1

elect sales_employee,fiscal_year,sale, rank() over (| Enter a SQL expression to filter resu

	sales_employee	fiscal_year	sale	ranknum
1	John	2,018	250	1
2	Bob	2,018	200	2
3	Alice	2,018	200	2
4	John	2,016	200	2
5	Alice	2,016	150	5
6	John	2,017	150	5
7	Bob	2,017	150	5
8	Alice	2,017	100	8
9	Bob	2,016	100	8

Difference between rank, dense_rank and row_number

Row number assigns a sequential integer to every row within its partition

Dense_rank assigns a rank to every row within its partition based on the ORDER BY clause. It assigns the same rank to the rows with equal values. If two or more rows have the same rank, then there will be no gaps in the sequence of ranked values.

Rank Similar to the DENSE_RANK() function except that there are gaps in the sequence of ranked values when two or more rows have the same rank.

Let's see the differences in below SQL query and its output.

```
select sales_employee,fiscal_year,sale,
 row_number () over(order by sale desc) as rownum,
 rank() over (order by sale desc ) as ranknum,
 dense_rank () over(order by sale desc) as denserank
 from sample_db.sales;
```

ales 1

elect sales_employee,fiscal_year,sale, row_numbe | Enter a SQL expression to filter results (use Ctrl+Space)

	sales_employee	fiscal_year	sale	rownum	ranknum	denserank
1	John	2,018	250	1	1	1
2	Bob	2,018	200	2	2	2
3	Alice	2,018	200	3	2	2
4	John	2,016	200	4	2	2
5	Alice	2,016	150	5	5	3
6	John	2,017	150	6	5	3
7	Bob	2,017	150	7	5	3
8	Alice	2,017	100	8	8	4
9	Bob	2,016	100	9	8	4

FIRST_VALUE()

Returns the value of the specified expression with respect to the first row in the window frame.

```
--first_value()
-- Returns the value of the specified expression with respect to the first row in the window frame.
-- write a query to display the highest paid job category of each country.
--

select *,
first_value (job_category) over(partition by country order by salary desc) as highest_paid_role
from sample_db.job_role jr ;
```

job_role 1

select *, first_value (job_category) over(partition by | Enter a SQL expression to filter results (use Ctrl+Space)

	id	job_category	country	salary	highest_paid_role
1	1	Data Scientist	India	80,000	Data Scientist
2	3	Developer	India	70,000	Data Scientist
3	4	Data Analyst	India	65,000	Data Scientist
4	5	Data Analyst	India	60,000	Data Scientist
5	18	ML Engineer	India	55,000	Data Scientist
6	17	Developer	India	50,000	Data Scientist
7	20	Developer	India	40,000	Data Scientist
8	10	Data Scientist	UK	160,000	Data Scientist
9	14	Developer	UK	151,000	Data Scientist
10	11	Data Analyst	UK	150,000	Data Scientist
11	9	Data Scientist	UK	140,000	Data Scientist
12	15	Data Analyst	UK	101,000	Data Scientist
13	16	Developer	UK	99,000	Data Scientist
14	13	Data Scientist	US	300,000	Data Scientist
15	12	Data Scientist	US	200,000	Data Scientist

Above query displays the highest paid job category of each country.

LAST_VALUE()

Returns the value of the specified expression with respect to the last row in the window frame.

Below query displays the lowest paid job category of each country.

```sql
-- last_value()
-- Returns the value of the specified expression with respect to the last row in the window frame.
-- write a query to display the lowest salary of each country.
select *,
last_value (job_category)
over(
partition by country
order by salary desc
range between unbounded preceding and unbounded following
)
as least_paid_role
from sample_db.job_role jr ;
```

ob_role 1

select *, last_value (job_category) over(partition b | Enter a SQL expression to filter results (use Ctrl+Space)

	id	job_category	country	salary	least_paid_role
1	1	Data Scientist	India	80,000	Developer
2	3	Developer	India	70,000	Developer
3	4	Data Analyst	India	65,000	Developer
4	5	Data Analyst	India	60,000	Developer
5	18	ML Engineer	India	55,000	Developer
6	17	Developer	India	50,000	Developer
7	20	Developer	India	40,000	Developer
8	10	Data Scientist	UK	160,000	Developer
9	14	Developer	UK	151,000	Developer
10	11	Data Analyst	UK	150,000	Developer
11	9	Data Scientist	UK	140,000	Developer
12	15	Data Analyst	UK	101,000	Developer
13	16	Developer	UK	99,000	Developer
14	13	Data Scientist	US	300,000	Data Analyst
15	12	Data Scientist	US	200,000	Data Analyst

Note: For LAST_VALUE() we need to change the default frame value of the over clause.

The default value is ***RANGE UNBOUNDED PRECEDING AND CURRENT ROW.***

NTH_VALUE()

It returns value of argument from Nth row of the window frame.

```sql
-- nth_value()
-- Returns value of argument from Nth row of the window frame
select *,
nth_value (job_category,4) over(partition by country order by salary desc) as nth_paid_role
from sample_db.job_role jr ;
```

job_role 1

select *, nth_value (job_category,4) over(partition | Enter a SQL expression to filter results (use Ctrl+Space)

	id	job_category	country	salary	nth_paid_role
1	1	Data Scientist	India	80,000	[NULL]
2	3	Developer	India	70,000	[NULL]
3	4	Data Analyst	India	65,000	[NULL]
4	5	Data Analyst	India	60,000	Data Analyst
5	18	ML Engineer	India	55,000	Data Analyst
6	17	Developer	India	50,000	Data Analyst
7	20	Developer	India	40,000	Data Analyst
8	10	Data Scientist	UK	160,000	[NULL]
9	14	Developer	UK	151,000	[NULL]
10	11	Data Analyst	UK	150,000	[NULL]
11	9	Data Scientist	UK	140,000	Data Scientist
12	15	Data Analyst	UK	101,000	Data Scientist
13	16	Developer	UK	99,000	Data Scientist
14	13	Data Scientist	US	300,000	[NULL]
15	12	Data Scientist	US	200,000	[NULL]

NTILE()

NTILE() distributes the rows for each window partition into a specified number of ranked groups.

Below query segregates all employess as high, mid and low paying professional.

Sal_bucket = 1 ➔ High paying Professional

Sal_bucket = 2 ➔ Mid paying Professional

Sal_bucket = 3 ➔ Low paying Professional

```
select *,
ntile (3) over( order by salary desc) as sal_bucket
from sample_db.job_role jr ;
```

job_role 1

select *, ntile (3) over(order by salary desc) as sal_b | Enter a SQL expression to filter results (use Ctrl+S

	id	job_category	country	salary	sal_bucket
1	13	Data Scientist	US	300,000	1
2	12	Data Scientist	US	200,000	1
3	10	Data Scientist	UK	160,000	1
4	14	Developer	UK	151,000	1
5	7	Data Scientist	US	150,000	1
6	11	Data Analyst	UK	150,000	1
7	9	Data Scientist	UK	140,000	1
8	19	ML Engineer	US	125,000	2
9	2	ML Engineer	US	120,000	2
10	6	Developer	US	110,000	2
11	15	Data Analyst	UK	101,000	2
12	8	Data Analyst	US	100,000	2
13	16	Developer	UK	99,000	2
14	1	Data Scientist	India	80,000	2
15	3	Developer	India	70,000	3
16	4	Data Analyst	India	65,000	3
17	5	Data Analyst	India	60,000	3
18	18	ML Engineer	India	55,000	3
19	17	Developer	India	50,000	3
20	20	Developer	India	40,000	3

PERCENT_RANK()

It calculates the percentile rank of a row in a partition or result set.

```
-- percent_rank()
-- Calculates the percentile rank of a row in a partition or result set
select *,
percent_rank() over( order by salary asc) as perc_ranking
from sample_db.job_role jr ;
```

b_role 1

elect *, percent_rank() over(order by salary asc) a | Enter a SQL expression to filter results (use Ctrl+S

	id	job_category	country	salary	perc_ranking
1	20	Developer	India	40,000	0
2	17	Developer	India	50,000	0.0526315789
3	18	ML Engineer	India	55,000	0.1052631579
4	5	Data Analyst	India	60,000	0.1578947368
5	4	Data Analyst	India	65,000	0.2105263158
6	3	Developer	India	70,000	0.2631578947
7	1	Data Scientist	India	80,000	0.3157894737
8	16	Developer	UK	99,000	0.3684210526
9	8	Data Analyst	US	100,000	0.4210526316
10	15	Data Analyst	UK	101,000	0.4736842105
11	6	Developer	US	110,000	0.5263157895
12	2	ML Engineer	US	120,000	0.5789473684
13	19	ML Engineer	US	125,000	0.6315789474
14	9	Data Scientist	UK	140,000	0.6842105263
15	7	Data Scientist	US	150,000	0.7368421053
16	11	Data Analyst	UK	150,000	0.7368421053
17	14	Developer	UK	151,000	0.8421052632
18	10	Data Scientist	UK	160,000	0.8947368421
19	12	Data Scientist	US	200,000	0.9473684211
20	13	Data Scientist	US	300,000	1

This is clearly not the end here. I have listed some widely used Window functions here. And this will solve the purpose in most of the cases.