What are Window Functions in SQL?

Window functions perform calculations on a set of rows that are related together. But, unlike the aggregate functions, windowing functions do not collapse the result of the rows into a single value. Instead, all the rows maintain their original identity and the calculated result is returned for every row.

OVER Clause

The OVER clause signifies a window of rows over which a window function is applied. It can be used with aggregate functions, like we have used with the SUM function here, thereby turning it into a window function.

```
select *, sum(cost_to_customer_per_qty) OVER() AS Total_cost
 from customer_purchases
```

Windowing with PARTITION BY

The PARTITION BY clause is used in conjunction with the OVER clause. It breaks up the rows into different partitions. These partitions are then acted upon by the window function.

```
select *,
cost_to_customer_per_qty*quantity,sum(cost_to_customer_per_qty*quantity)
OVER(partition by vendor_id)
AS Total_cost
 from customer_purchases
```

1. Row_Number

Sometimes your dataset might not have a column depicting the sequential order of the rows.

ROW_NUMBER function is a SQL ranking function that **assigns a sequential rank number to each new record in a partition**. When the SQL Server ROW NUMBER

function detects two identical values in the same partition, it assigns different rank numbers to both.

```
SELECT
	ROW_NUMBER() OVER (
		ORDER BY product_name
	) row_num,
   product_name,
   product_size
FROM
	product
ORDER BY
	product_name;
```

Rank:

- The RANK() function is a window function that could be used in SQL Server to calculate a rank for each row within a partition of a result set.

- The same rank is assigned to the rows in a partition which have the same values.
- The rank of the first row is 1.
- The ranks may not be consecutive in the RANK() function as it adds the number of repeated rows to the repeated rank to calculate the rank of the next row.

```
SELECT
	rank() OVER (
		ORDER BY product_name
	) rank_num,
   product_name,
   product_size
FROM
	product
ORDER BY
	product_name;
```

```
insert into product values (24,'Carrots','sold by weight',1,'lbs')
```

https://www.mysqltutorial.org/mysql-window-functions/

DENSE_RANK

- It assigns rank to each row within the partition.
- Just like the rank function, the first row is assigned rank 1 and rows having the same value have the same rank.
- The difference between RANK() and DENSE_RANK() is that in DENSE_RANK(), for the next rank after two of the same rank, consecutive integers are used, no rank is skipped.

Example of ROWS?RANGE preceding :

ROWS: means specified row or set of rows

RANGE: refers to those same rows plus any other that have same matching values

```
create table demo_window_function
(
cid int
);


insert into demo_window_function select 1;
insert into demo_window_function select 2;
insert into demo_window_function select 3;
insert into demo_window_function select 4;
insert into demo_window_function select 5;
insert into demo_window_function select 6;
insert into demo_window_function select 7;
insert into demo_window_function select 8;
insert into demo_window_function select 9;
insert into demo_window_function select 10;

select * from demo_window_function;
```

#Case1: To read previous row value

```
select
	cid,
	sum(cid) over(
	order by cid rows between 1 preceding and 1 preceding) as cid
from
	demo_window_function;
```

#Case2: To read all the previous rows

```
select
	cid,
	sum(cid) over(
	order by cid rows between UNBOUNDED preceding and 1 preceding) as
cid2
from
	demo_window_function;
```

#Case3: To read previous 2 rows only

```
select
	cid,
	sum(cid) over(
	order by cid rows between 2 preceding and 1 preceding) as cid3
from
	demo_window_function;
```

#Case4: To read the next row only

```
select
	cid,
	sum(cid) over(
	order by sid rows between 1 following and 1 following) as cid3
from
	demo_window_function;
```

#Case5: To read all the following rows

```
select
	cid,
	sum(cid) over(
	order by cid rows between 1 following and unbounded following) as cid3
from
	demo_window_function;

#Case6: To read next 2 rows only
select
	cid,
	sum(cid) over(
	order by cid rows between 1 following and 2 following) as cid3
from
	demo_window_function;

#Case7: If you want to include current row
select
	cid,
	sum(cid) over(
	order by cid rows between current row and 2 following) as cid3
from
	demo_window_function;

#Case8: Calculate sum of all rows till current row
select
	cid,
	sum(cid) over(
	order by cid rows between UNBOUNDED preceding and current row) as
cid3
from
	demo_window_function;
```

NTILE Functions:

- NTILE() function in SQL Server is a window function that distributes rows of an ordered partition into a pre-defined number of roughly equal groups.
- It assigns each group a number_expression ranging from 1.
- NTILE() function assigns a number_expression for every row in a group, to which the row belongs.
- When number of rows isn't divisible by the number_expression, the NTILE() function results the groups of two sizes with the difference by one

Parameters of syntax in detail :

- **number_expression**

  The number_expression is the integer into which the rows are divided.

- **PARTITION BY clause**

  The PARTITION BY is optional, it differs the rows of a result set into partitions where the NTILE() function is used.

- **ORDER BY clause**

  The ORDER BY clause defines the order of rows in each partition where the NTILE() is used.

SELECT

```
        product_id,
        product_name,
        product_category_id,
        NTILE(5) OVER (ORDER BY product_name DESC) AS price_ntile
FROM farmers_market.product
ORDER BY product_name DESC
```

Inside product_name

```
SELECT
        product_id,
        product_name,
        product_category_id,
        NTILE(2) OVER (ORDER BY product_name DESC) AS price_ntile,
    NTILE(2) OVER (PARTITION BY product_name ORDER BY product_name
DESC) AS product_tile
FROM farmers_market.product
ORDER BY product_name DESC
```

LAG() window function

- At many instances, users would like to access data of the previous row or any row before the previous row from the current row.

- To solve this problem SQL Server's LAG() window function can be used.
- SQL Server provides a LAG() function which is very useful in case the current row values need to be compared with the data/value of the previous record or any record before the previous record.

Syntax :

```
LAG (scalar_expression [, offset] [, default])
OVER ( [ partition_by ] order_by )
```

Where :

1. **scalar_expression –**

   The value to be returned based on the specified offset.

2. **offset –**

   The number of rows back from the current row from which to obtain a value. If not specified, the default is 1.

3. **default –**

   default is the value to be returned if offset goes beyond the scope of the partition. If a default value is not specified, NULL is returned.

4. **over ([partition_by] order_by) –**

   partition_by divides the result set produced by the FROM clause into partitions to which the function is applied. If you omit the PARTITION BY clause, the function treats the whole result set as a single group. By default order_by clause sorts in ascending order.

3) Show the employee details and first_name who have the fifth-highest salary in each job category.

```
select
* from
(
select *,
ROW_NUMBER() OVER(partition by job_id order by salary) as ans
```

```
from employees
) x
where x.ans = 5
```

6) Write a Query to find the first day of the most recent job of every employee. Return the first_name of the employee and the new recent job as 'recent_job'.

```
select

distinct first_name,start_date

FROM

(

SELECT *, last_value(start_date)

over (partition by jhist.employee_id order by start_date asc)

as 'recent_job',

rank()

over (partition by jhist.employee_id order by start_date desc)

as 'rank_job'

 from job_history  jhist

) x,employees

WHERE x.employee_id=employees.employee_id

AND x.rank_job = 1

order by x.start_date;
```

All query elements are processed in a very strict order:

- **FROM** - the database gets the data from tables in FROM clause and if necessary performs the JOINs,
- **WHERE** - the data are filtered with conditions specified in WHERE clause,
- **GROUP BY** - the data are grouped by with conditions specified in WHERE clause,
- **Aggregate functions** - the aggregate functions are applied to the groups created in the GROUP BY phase,
- **HAVING** - the groups are filtered with the given condition,
- **Window functions,**
- **SELECT** - the database selects the given columns,
- **DISTINCT** - repeated values are removed,
- **UNION/INTERSECT/EXCEPT** - the database applies set operations,
- **ORDER BY** - the results are sorted,
- **OFFSET** - the first rows are skipped,
- **LIMIT/FETCH/TOP** - only the first rows are selected

Date And Time Function

DATE_ADD() function in MySQL is used to add a specified time or date interval to a specified date and then return the date.

Syntax:

```
DATE_ADD(date, INTERVAL value addunit)
```

Parameter: This function accepts two parameters which are illustrated below:

- **date** –

  Specified date to be modified.
- **value addunit** –

  Here the value is the date or time interval to add. This value can be both positive and negative. And here the addunit is the type of interval to add such as SECOND, MINUTE, HOUR, DAY, YEAR, MONTH, etc.

Example :

```
SELECT DATE_ADD("2017-11-22", INTERVAL 3 YEAR);

SELECT DATE_ADD("2020-9-22", INTERVAL 2 MONTH);

SELECT DATE_ADD("2020-11-12", INTERVAL 10 DAY);
```

```
SELECT DATE_ADD("2020-11-22 06:12:10", INTERVAL 3 HOUR);

SELECT DATE_ADD("2020-11-22 09:06:10", INTERVAL 3 MINUTE);

SELECT DATE_ADD("2020-11-22 09:09:5", INTERVAL 5 SECOND);
```

SQL CTE

In the simplest terms, CTEs allow you to create temporary datasets that you can reference later on in a query.

Those temporary datasets are “available” to use for the duration of the query itself, but they are not stored in your database so they are gone once your query has been executed.

Why do we need them?

At their core CTEs do two things:

1. They solve what I like to call the “logic on top of logic” problem. This occurs when you have to perform data manipulation and then use the resulting dataset to perform more manipulation.
2. They make your code more readable and easier to work with

Example:

-- QUERY 1 :

Question : Fetch employees who earn more than avg sal of all

employees?

Ans :

```
with avg_sal(avg_salary) as
                (select ROUND(avg(salary),0) FROM emp)
select *
from emp e
join avg_sal av on e.salary > av.avg_salary
```

Query 2:
Find stores who's sales were better than avg sales across all stores?

```
-- Find total sales per each store
select s.store_id, sum(s.cost) as total_sales_per_store
from sales s
group by s.store_id;


-- Find average sales with respect to all stores
select ROUND(avg(total_sales_per_store),0) avg_sale_for_all_store
from (select s.store_id, sum(s.cost) as total_sales_per_store
        from sales s
        group by s.store_id) x;



-- Find stores who's sales where better than the average sales across all
stores
select *
from   (select s.store_id, sum(s.cost) as total_sales_per_store
                            from sales s
```

```
                        group by s.store_id
        ) total_sales
join   (select ROUND(avg(total_sales_per_store),2)
avg_sale_for_all_store
                        from (select s.store_id, sum(s.cost) as
total_sales_per_store
                                from sales s
                                        group by s.store_id) x
        ) avg_sales
on total_sales.total_sales_per_store >
avg_sales.avg_sale_for_all_store;


-- Using WITH clause
WITH total_sales as
                (select s.store_id, sum(s.cost) as total_sales_per_store
                from sales s
                group by s.store_id),
        avg_sales as
                (select ROUND(avg(total_sales_per_store),0)
avg_sale_for_all_store
                from total_sales)
select *
from   total_sales
join   avg_sales
on total_sales.total_sales_per_store >
avg_sales.avg_sale_for_all_store;
```

Views

The key thing to remember about SQL views is that, in contrast to a CTE, **a view is a physical object in a database and is stored on a disk**. However, **views store the query only, not the data returned by the query**. The data is computed each time you reference the view in your query.

64

View Vs Materialized View

A view uses a query to pull data from the underlying tables.

A materialized view is a table on disk that contains the result set of a query.

Materialized views are primarily used to increase application performance when it isn't feasible or desirable to use a standard view with indexes applied to it