Advance Analysis with windows function

- In SQL, **window functions** are a powerful feature used to perform calculations across a set of table rows that are related to the current row.
- Unlike aggregate functions, window functions do not collapse the result set but instead allow you to perform calculations while keeping the individual rows.

Key Types of Window Functions:

1. **Ranking Functions**: Assign ranks or other numeric values to rows.
2. **Aggregate Window Functions**: Apply aggregate functions like `SUM`, `AVG`, etc., over a window of rows.
3. **Analytic Functions**: Perform analysis across a set of rows, like calculating running totals, moving averages, etc.

```
SELECT
  column_name,
  window_function() OVER ([PARTITION BY column_name] [ORDER BY column_name]) AS alias_name
FROM table_name;
```

Example : Windows Function

```
select BusinessEntityID,TerritoryID,
SalesQuota,Bonus,CommissionPct,
sum(SalesYTD) over(partition by businessentityid) as Total_YTD,
max(SalesYTD) over() as Max_YTD,
SalesYTD*100/max(SalesYTD) over() as '% of best performer'
from Sales.SalesPerson
order by '% of best performer' DESC
```

Results | Messages

	BusinessEntityID	TerritoryID	SalesQuota	Bonus	CommissionPct	Total_YTD	Max_YTD	% of best performer
1	276	4	250000.00	2000.00	0.015	4251368.5497	4251368.5497	100.00
2	289	10	250000.00	5150.00	0.02	4116871.2277	4251368.5497	96.8363
3	275	2	300000.00	4100.00	0.012	3763178.1787	4251368.5497	88.5168
4	277	3	250000.00	2500.00	0.015	3189418.3662	4251368.5497	75.0209
5	290	7	250000.00	985.00	0.016	3121616.3202	4251368.5497	73.4261
6	282	6	250000.00	5000.00	0.015	2604540.7172	4251368.5497	61.2635
7	281	4	250000.00	3550.00	0.01	2458535.6169	4251368.5497	57.8292
8	279	5	300000.00	6700.00	0.01	2315185.611	4251368.5497	54.4574
9	288	8	250000.00	75.00	0.018	1827066.7118	4251368.5497	42.9759
10	284	1	300000.00	3900.00	0.019	1576562.1966	4251368.5497	37.0836
11	283	1	250000.00	3500.00	0.012	1573012.9383	4251368.5497	37.0001
12	278	6	250000.00	500.00	0.01	1453719.4653	4251368.5497	34.1941
13	286	9	250000.00	5650.00	0.018	1421810.9242	4251368.5497	33.4436
14	280	1	250000.00	5000.00	0.01	1352577.1325	4251368.5497	31.815
15	274	NULL	NULL	0.00	0.00	559697.5639	4251368.5497	13.1651
16	287	NULL	NULL	0.00	0.00	519905.932	4251368.5497	12.2291

- Here instead of collapsing rows windows function added a new column for us to do calculation e.g. Aggregare functions like `sum()/max()` will simply sum all and collapse every row....but in `windows function sum() or max() are in a separate columns`

OVER()

In SQL, the `OVER()` clause is used in conjunction with **window functions** to define the window or set of rows over which the function operates. It allows you to apply aggregate, ranking, and analytic functions across a specific window of rows without collapsing the result set, as would happen with a regular `GROUP BY` clause.

Partition by()

- In SQL, the **`PARTITION BY`** clause is used with **window functions** to divide the result set into partitions (or subsets of rows).
- The window function is then applied independently to each partition.
- It's similar to a **`GROUP BY`** clause but doesn't collapse the rows; instead, it allows window functions to work on individual rows within each partition.

ROW_NUMBER()

In SQL, the **`ROW_NUMBER()`** function is a window function that assigns a unique sequential integer to rows within a partition of a result set, starting from 1 for the first row in each partition. It's commonly used for tasks like pagination, ranking, or identifying duplicates.

```
ROW_NUMBER() OVER ([PARTITION BY partition_expression] ORDER BY order_expression)
```

Components of row_number():

1. **`PARTITION BY`** (Optional): Divides the result set into partitions. `ROW_NUMBER()` is applied separately to each partition.
2. **`ORDER BY`** (Required): Specifies the order in which the rows are assigned their row number within the partition.
   - Means we must always use `ORDER BY`

```
select BusinessEntityID,TerritoryID,
SalesQuota,Bonus,CommissionPct,salesytd,
sum(SalesYTD) over(partition by businessentityid) as Total_YTD,
max(SalesYTD) over() as Max_YTD,
SalesYTD*100/max(SalesYTD) over() as '% of best performer',
ROW_NUMBER() over (ORDER by salesytd DESC ) as sales_rank
from Sales.SalesPerson
order by '% of best performer' DESC
```

Results | Messages

	BusinessEntityID	TerritoryID	SalesQuota	Bonus	CommissionPct	salesytd	Total_YTD	Max_YTD	% of best performer	sales_rank
1	276	4	250000.00	2000.00	0.015	4251368.5497	4251368.5497	4251368.5497	100.00	1
2	289	10	250000.00	5150.00	0.02	4116871.2277	4116871.2277	4251368.5497	96.8363	2
3	275	2	300000.00	4100.00	0.012	3763178.1787	3763178.1787	4251368.5497	88.5168	3
4	277	3	250000.00	2500.00	0.015	3189418.3662	3189418.3662	4251368.5497	75.0209	4
5	290	7	250000.00	985.00	0.016	3121616.3202	3121616.3202	4251368.5497	73.4261	5
6	282	6	250000.00	5000.00	0.015	2604540.7172	2604540.7172	4251368.5497	61.2635	6
7	281	4	250000.00	3550.00	0.01	2458535.6169	2458535.6169	4251368.5497	57.8292	7
8	279	5	300000.00	6700.00	0.01	2315185.611	2315185.611	4251368.5497	54.4574	8
9	288	8	250000.00	75.00	0.018	1827066.7118	1827066.7118	4251368.5497	42.9759	9
10	284	1	300000.00	3900.00	0.019	1576562.1966	1576562.1966	4251368.5497	37.0836	10
11	283	1	250000.00	3500.00	0.012	1573012.9383	1573012.9383	4251368.5497	37.0001	11
12	278	6	250000.00	500.00	0.01	1453719.4653	1453719.4653	4251368.5497	34.1941	12
13	286	9	250000.00	5650.00	0.018	1421810.9242	1421810.9242	4251368.5497	33.4436	13
14	280	1	250000.00	5000.00	0.01	1352577.1325	1352577.1325	4251368.5497	31.815	14
15	274	NULL	NULL	0.00	0.00	559697.5639	559697.5639	4251368.5497	13.1651	15

RANK() vs DENSE_RANK()

Here's a comparison between `RANK()` and `DENSE_RANK()` in table form:

Aspect	`RANK()`	`DENSE_RANK()`
Ranking Order	Assigns a rank to each row in the result set.	Assigns a rank to each row in the result set.
Ties Handling	Tied rows get the same rank.	Tied rows get the same rank.
Gaps in Ranking	Leaves gaps in ranks when there are ties.	No gaps; consecutive ranking even with ties.
Next Rank After Tie	The next rank skips by the number of ties.	The next rank is sequential without skipping numbers.
Example (Ties)	If two rows are ranked 2, the next row will be ranked 4.	If two rows are ranked 2, the next row will be ranked 3.
Use Case	Use when gaps in ranking are acceptable or desired.	Use when consecutive ranking without gaps is preferred.

Example Data:

EmployeeID	SalesAmount
1	5000
2	4000
3	4000
4	3000

Result with `RANK()`:

```
SELECT
  EmployeeID,
  SalesAmount,
  RANK() OVER (ORDER BY SalesAmount DESC) AS Rank
FROM Sales;
```

EmployeeID	SalesAmount	Rank
1	5000	1
2	4000	2
3	4000	2
4	3000	4

Result with `DENSE_RANK()`:

```
SELECT
  EmployeeID,
  SalesAmount,
```

```
  DENSE_RANK() OVER (ORDER BY SalesAmount DESC) AS DenseRank
FROM Sales;
```

EmployeeID	SalesAmount	DenseRank
1	5000	1
2	4000	2
3	4000	2
4	3000	3

Example: Rank, Dense_rank

```
select BusinessEntityID,
SalesQuota,salesytd,
sum(SalesYTD) over(partition by businessentityid) as Total_YTD,
max(SalesYTD) over() as Max_YTD,
SalesYTD*100/max(SalesYTD) over() as '% of best performer',
ROW_NUMBER() over (ORDER by SalesQuota DESC ) as sales_rank,
RANK() OVER (ORDER BY SalesQuota DESC) as 'rank()',
DENSE_RANK() OVER (ORDER by SalesQuota DESC) as 'DenseRank()'
from Sales.SalesPerson
order by SalesQuota DESC
```

Results Messages

	BusinessEntityID	SalesQuota	salesytd	Total_YTD	Max_YTD	% of best performer	sales_rank	rank()	DenseRank()
1	275	300000.00	3763178.1787	3763178.1787	4251368.5497	88.5168	1	1	1
2	279	300000.00	2315185.611	2315185.611	4251368.5497	54.4574	2	1	1
3	284	300000.00	1576562.1966	1576562.1966	4251368.5497	37.0836	3	1	1
4	286	250000.00	1421810.9242	1421810.9242	4251368.5497	33.4436	4	4	2
5	288	250000.00	1827066.7118	1827066.7118	4251368.5497	42.9759	5	4	2
6	289	250000.00	4116871.2277	4116871.2277	4251368.5497	96.8363	6	4	2
7	290	250000.00	3121616.3202	3121616.3202	4251368.5497	73.4261	7	4	2
8	280	250000.00	1352577.1325	1352577.1325	4251368.5497	31.815	8	4	2
9	281	250000.00	2458535.6169	2458535.6169	4251368.5497	57.8292	9	4	2
10	282	250000.00	2604540.7172	2604540.7172	4251368.5497	61.2635	10	4	2
11	283	250000.00	1573012.9383	1573012.9383	4251368.5497	37.0001	11	4	2
12	276	250000.00	4251368.5497	4251368.5497	4251368.5497	100.00	12	4	2
13	277	250000.00	3189418.3662	3189418.3662	4251368.5497	75.0209	13	4	2
14	278	250000.00	1453719.4653	1453719.4653	4251368.5497	34.1941	14	4	2
15	274	NULL	559697.5639	559697.5639	4251368.5497	13.1651	15	15	3

LEAD() and LAG()

In SQL, **`LEAD()`** and **`LAG()`** are window functions that allow you to access data from subsequent (next) or preceding (previous) rows within the result set without needing to join the table to itself. These functions are commonly used in time series or sequential data analysis.

- **`LEAD()`** gives access to the value of a subsequent `(next)` row from the current row. It allows you to "look forward."
- **`LAG()`** gives access to the value of a preceding `(previous)` row from the current row. It allows you to "look back."

```
LEAD(column_name, offset, default_value) OVER (PARTITION BY partition_column ORDER BY order_column)
```

```sql
SELECT
    EmployeeID,
    SalesAmount,
    LAG(SalesAmount, 1) OVER (ORDER BY SalesAmount DESC) AS Previous_SalesAmount,
    LEAD(SalesAmount, 1) OVER (ORDER BY SalesAmount DESC) AS Next_SalesAmount
FROM Sales;
```

Example with Data:

EmployeeID	SalesAmount
1	5000
2	4000
3	3000
4	2000

Result:

EmployeeID	SalesAmount	Previous_SalesAmount	Next_SalesAmount
1	5000	NULL	4000
2	4000	5000	3000
3	3000	4000	2000
4	2000	3000	NULL

- Note: we can use dafault value to fill the NULL

```sql
select SalesOrderID,OrderDate,
CustomerID,TotalDue,
lead(TotalDue,1) OVER (partition by customerid order by salesorderid) as Next_Due,
lag (TotalDue,1) OVER (partition by customerid order by salesorderid) as Prev_Due
from sales.salesorderheader
order by CustomerID,SalesOrderID
```

Results | Messages

	SalesOrderID	OrderDate	CustomerID	TotalDue	Next_Due	Prev_Due
1	43793	2011-06-21 00:00:00.000	11000	3756.989	2587.8769	NULL
2	51522	2013-06-20 00:00:00.000	11000	2587.8769	2770.2682	3756.989
3	57418	2013-10-03 00:00:00.000	11000	2770.2682	NULL	2587.8769
4	43767	2011-06-17 00:00:00.000	11001	3729.364	2674.0227	NULL
5	51493	2013-06-18 00:00:00.000	11001	2674.0227	650.8008	3729.364
6	72773	2014-05-12 00:00:00.000	11001	650.8008	NULL	2674.0227
7	43736	2011-06-09 00:00:00.000	11002	3756.989	2535.964	NULL
8	51238	2013-06-02 00:00:00.000	11002	2535.964	2673.0613	3756.989
9	53237	2013-07-26 00:00:00.000	11002	2673.0613	NULL	2535.964
10	43701	2011-05-31 00:00:00.000	11003	3756.989	2562.4508	NULL
11	51315	2013-06-07 00:00:00.000	11003	2562.4508	2674.4757	3756.989
12	57783	2013-10-10 00:00:00.000	11003	2674.4757	NULL	2562.4508
13	43810	2011-06-25 00:00:00.000	11004	3756.989	2626.5408	NULL
14	51595	2013-06-24 00:00:00.000	11004	2626.5408	2673.0613	3756.989
15	57293	2013-10-01 00:00:00.000	11004	2673.0613	NULL	2626.5408
16	43704	2011-06-01 00:00:00.000	11005	3729.364	2610.3084	NULL

FIRST_VALUE()

In SQL, `FIRST_VALUE()` is a window function that returns the first value in an ordered partition of a result set. It allows you to retrieve the first value within a group of rows, which is useful for comparisons or when you need to fetch the initial record in a sorted group.

```
FIRST_VALUE(column_name) OVER (PARTITION BY partition_column ORDER BY order_column)
```

Example: FIRST_VALUE()

```
select SalesOrderID,OrderDate,
CustomerID,TotalDue,
row_number() over(partition by customerid order by salesorderid) as rank,
first_value(TotalDue) over (partition by customerid order by salesorderid) as first from
sales.salesorderheader
order by CustomerID,SalesOrderID
```

Results | Messages

	SalesOrderID	OrderDate	CustomerID	TotalDue	rank	first
1	43793	2011-06-21 00:00:00.000	11000	3756.989	1	3756.989
2	51522	2013-06-20 00:00:00.000	11000	2587.8769	2	3756.989
3	57418	2013-10-03 00:00:00.000	11000	2770.2682	3	3756.989
4	43767	2011-06-17 00:00:00.000	11001	3729.364	1	3729.364
5	51493	2013-06-18 00:00:00.000	11001	2674.0227	2	3729.364
6	72773	2014-05-12 00:00:00.000	11001	650.8008	3	3729.364
7	43736	2011-06-09 00:00:00.000	11002	3756.989	1	3756.989
8	51238	2013-06-02 00:00:00.000	11002	2535.964	2	3756.989
9	53237	2013-07-26 00:00:00.000	11002	2673.0613	3	3756.989
10	43701	2011-05-31 00:00:00.000	11003	3756.989	1	3756.989
11	51315	2013-06-07 00:00:00.000	11003	2562.4508	2	3756.989
12	57783	2013-10-10 00:00:00.000	11003	2674.4757	3	3756.989
13	43810	2011-06-25 00:00:00.000	11004	3756.989	1	3756.989
14	51595	2013-06-24 00:00:00.000	11004	2626.5408	2	3756.989
15	57293	2013-10-01 00:00:00.000	11004	2673.0613	3	3756.989

The Query display the first value of each group [CustomerID]

ROWS_BETWEEN()

In SQL, the `ROWS BETWEEN` clause is often used in conjunction with window functions to define a specific range of rows to consider for calculations. It is part of the `OVER()` clause and allows you to specify a frame of rows relative to the current row.

```
<window_function> OVER (
    [PARTITION BY partition_column]
    ORDER BY order_column
    ROWS BETWEEN <start> AND <end>
)
```

Frame Specifications:

- **`UNBOUNDED PRECEDING`** : Refers to the start of the partition.
- **`N PRECEDING`** : Refers to the Nth row before the current row.
- **`CURRENT ROW`** : Refers to the current row.
- **`N FOLLOWING`** : Refers to the Nth row after the current row.
- **`UNBOUNDED FOLLOWING`** : Refers to the end of the partition.

```
SELECT
    OrderDate,
    SumTotalDue,
    AVG(SumTotalDue) OVER (ORDER BY OrderDate ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS day_rolling
FROM (
    SELECT
        OrderDate,
        SUM(TotalDue) AS SumTotalDue
    FROM Sales.SalesOrderHeader
    GROUP BY OrderDate
) temp
ORDER BY OrderDate;
-- 3 day rolling average including current day
```

Results | Messages

	OrderDate	SumTotalDue	day_rolling
1	2011-05-31 00:00:00.000	567020.9498	567020.9498
2	2011-06-01 00:00:00.000	15394.3298	291207.6398
3	2011-06-02 00:00:00.000	16588.4572	199667.9122
4	2011-06-03 00:00:00.000	7907.9768	13296.9212
5	2011-06-04 00:00:00.000	16588.4572	13694.9637
6	2011-06-05 00:00:00.000	15815.9536	13437.4625
7	2011-06-06 00:00:00.000	8680.4804	13694.9637