CASE STATEMENT

- Evaluates a list of condition and returns a value when the first condition is met.

Syntax.

CASE ← Start of logic

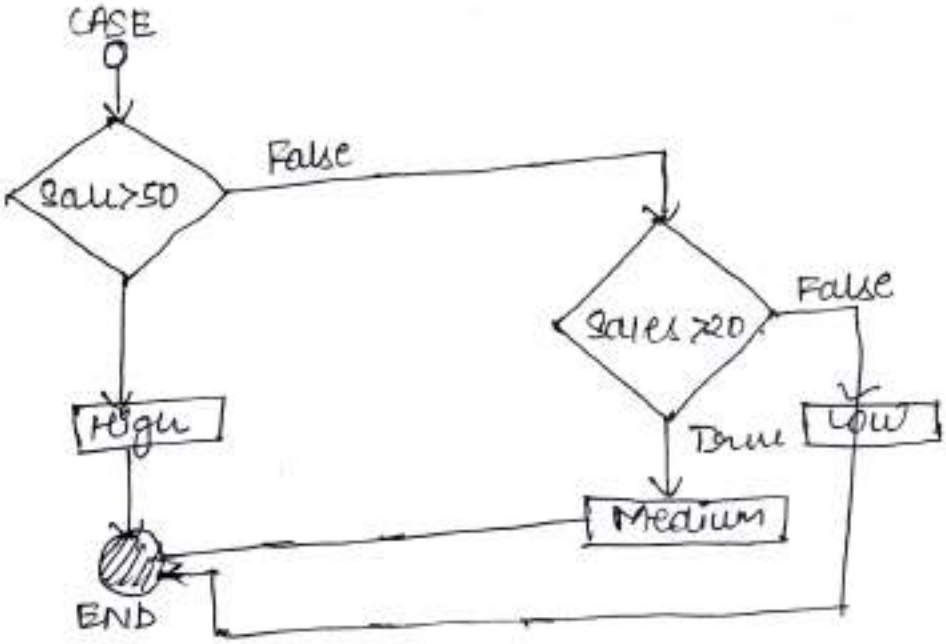
Condition to be evaluated ← WHEN condition1 THEN result1 ← Result of condition is True

WHEN condition 2 THEN result2

Default value (optional) ← ELSE Result

If none of the when conditions are True.

END → End of logic




```
CASE
   WHEN Sale > 50 THEN 'HIGH'
   WHEN Sales >20 THEN 'Medium'
   ELSE 'LOW'
END
```

USE CASE

1) CATEGORIZING DATA

- Main purpose of case statement is Data Transformation.

 # Drive New information

- Create new columns based on existing data
- Group data into different categories based on certain conditions.

Eg: Report for Total Sales for each category:

High If Sales > 50

Medium If Sales < 50 AND Sales > 20

Low If Sales <= 20

Sort Result from lowest to Highest.

```
SELECT
    category,
    SUM(Sales) AS TotalSales
    FROM (
        SELECT
        orderID,
        Sales,
        CASE WHEN Sales > 50 THEN 'High'
             WHEN Sales > 20 THEN 'Medium'
             ELSE 'Low'
        END Category
        FROM Sales.orders
    ) t
    GROUP BY category
```

CASE Statement :- Rules.

Data type of result Must be matching.

* Mapping Values

↓
Transform the values from one form to another.

```
SELECT
 EmployeeID,
  Firstname,
  Lastname,
  Gender,
  CASE
      WHEN Gender = 'F' THEN 'Female'
       WHEN Gender = 'M' THEN 'Male'
        ELSE 'NOT available'
   END GenderFullText
   FROM Sales.Employees.
```

Change M → Male

* CASE Statement - Quick Form

```
CASE
   WHEN country = 'Germany' THEN 'DE'
   WHEN country = 'India' THEN 'IN'
   WHEN country = 'United States' THEN 'US'
   WHEN country = 'France' THEN 'FR'
    ELSE 'n/a'
END
```

FULL FORM

use country multiple times

CASE Country → column Name (only one).

WHEN 'Germany' THEN 'DE'
WHEN 'India' THEN 'IN'
WHEN 'United States' THEN 'US'
ELSE 'n/a'

END

* Handling NULLS

- Replace NULLS with specific value.
- NULLS can lead to inaccurate results, which can lead to wrong decision-making.

```
SELECT
  CustomerID,
  LastName,
  Score,
  AVG(CASE
        WHEN Score IS NULL THEN 0
        ELSE Score
  END) OVER() AvgScore
FROM Sales.Customers
```

* Conditional aggregation – use case.

Apply aggregate functions only on subsets of data that fulfil certain conditions.

```
SELECT
  CustomerID,
  SUM(CASE
        WHEN Sales > 30 THEN 1
        ELSE 0
  END) TotalOrdersHighSales,
  COUNT(*) TotalOrders
FROM Sales.Orders
GROUP BY CustomerID
```

• aggregate functions.

```
SELECT
customer_id,
COUNT(*) AS total_order
SUM(sales) AS Total_Sales
AVG(sales) AS avg_sales
MAX(sales) AS Highest_sales
MIN(sales) AS lowest_sales
FROM orders
GROUP BY customer_id.
```

Score.

```
SELECT
CustomerID,
COUNT(*) AS Total_nr_Customer
SUM(CASE
    WHEN Score IS NULL THEN 0
    ELSE Score
END) AS Total_Score
AVG(CASE
    WHEN Score IS NULL THEN 0
    ELSE Score
END) AS avg_Score
FROM Sales.Customers
GROUP BY CustomerID.
```

WINDOWS FUNCTION

- perform calculations (e.g aggregation) on a specific subset of data, without losing the level of details of rows.

```
SELECT
    SUM(Sales) TotalSales
FROM Sales.orders
```

Total sales of order.

```
SELECT
    productID,
    SUM(Sales) Total Sales
FROM Sales.orders
GROUP BY productID
```

Total sales for each product

```
SELECT
    orderID,
    OrderDate,
    productID,
    SUM(Sales) Total Sales
FROM Sales.orders
GROUP BY productID
```

#Group By Rule:

- All columns in SELECT must be included in GROUP BY.
- Can't do aggregations and provide details at same time.

```
SELECT
    SUM(Sales) OVER()
FROM Sales.orders
```

Result Granularity

Window function return a result for each row.

```
SELECT
    SUM(Sales) OVER (PARTITION BY ProductID)
FROM Sales.orders
```

Total sales by each product.

```
SELECT
    OrderID,
    OrderDate,
    ProductID,
    SUM(Sales) OVER (PARTITION BY productID) AS
        TotalSalesByProduct
FROM Sales.orders
```

Window Function

Advanced Data Analytics (Aggregations + Details)

#THE SYNTAX.

OVER CLAUSE.

Window Function | partition clause | ORDER CLAUSE | FRAME CLAUSE

↓ perfor calculation within a window

Function Expression.

Arguments you pass to a function.

Avg(Sales)

Empty — Rank() OVER (order BY orderDate)

COLUMN — AVG(Sales) OVER (ORDER BY orderDate)

NUMBER — NTILE(2) OVER (ORDER BY orderDate)

Multiple Arguments — LEAD(Sales, 2, 10) OVER (ORDER BY orderDate)

Conditional Logic — SUM(CASE) (ORDER BY orderDate)

#OVER() CLAUSE : Tells SQL that the function used is a window function.

- Defines a window or subset of data.

#PARTITION BY Category

↓ partition clause

- Divides the dataset into windows (partition)
- Divides the result set into partitions (windows)

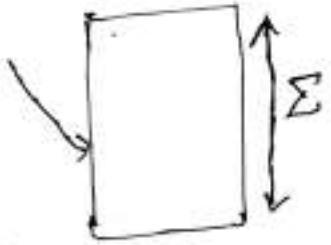
#partition clause

↙

divides the rows into groups, based on column/s.

SUM(sales) OVER()

calculation done on entire dataset.



↓

```
SELECT
    orderID,
    orderDate,
    SUM(Sales) OVER() TotalSales
FROM Sales.orders
```

SUM(sales) OVER (PARTITION BY product)

calculation done individually on each window.

```
SELECT
    orderID,
    orderDate,
    productID,
    SUM(Sales) OVER(
        PARTITION BY productID)
    TotalSales Each
FROM Sales.orders.
```

Flexibility of window

Allow aggregation of data at different granularities within same query.

#ORDER BY.

Sort the data within a window

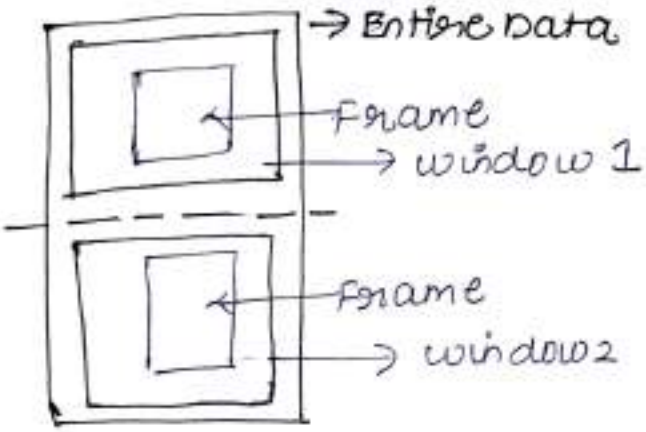
(ASC / DESC)

```
SELECT
    orderID,
    Sales,
    RANK() OVER(order BY Sales DESC)
FROM Sales.orders
```

WINDOW FRAME

Define a subset of rows in a window. (each) that is reliant for calculation.

ROW UNBOUNDED PRECEDING



AVG(Sales) OVER(PARTITION BY category ORDER BY orderdate

ROWS BETWEEN CURRENT ROW AND UNBOUNDED PRECEDING

ROWS → Frame types → ROWS, RANGE

FRAME BOUNDARY (lower value)
- CURRENT ROW
- N PRECEDING
- UNBOUNDED PRECEDING

FRAME BOUNDARY (Higher value)
- CURRENT ROW
- N FOLLOWING
- UNBOUNDED FOLLOWING

Rules:- Frame clause can only be used together with order BY clause.

- Lower value must be before the higher value

N-Following : The n-th row before the current row.

unbounded following : last possible ROW within a window.

n-PRECEDING : n-th row before the current ROW.

unbounded preceding : first possible ROW within a window.

compact frame.

For only preceding the current ROW can be skipped.

Shortform ROWS 2 PRECEDING

ROWS UNBOUNDED PRECEDING.

Default frame : SQL uses default Frame, If ORDER BY is used without FRAME.

↓

ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.

RULE.

1) Window Functions can be used ONLY in SELECT and ORDER BY CLAUSE.
2) Nesting Window Function NOT allowed.
3) SQL execute WINDOW Function after WHERE clause.
4) WINDOW function can be used together with group BY - in the same query, only If same columns are used.

Window Aggregate function:

AVG(Sales) OVER(PARTITION BY ProductID ORDER BY Sale)

- AVG(Sales) → Required (only Numeric)
- PARTITION BY ProductID, ORDER BY Sale → Both is optional

only count is accept all data types. And all function only accept Numeric.

COUNT() : Return the no. of rows within a window.

COUNT(Sales) : Count number of Non-NULL values in the column.

COUNT(1) = COUNT(*) Both are same.

[Count total number of rows including duplicates, not unique values.]

#1 use case :- (overall analysis) Quick summary or snapshot of entire dataset.

#2 use case :- (Total per groups) Group wise analysis, to understand within different categories.

```
SELECT
    orderID,
    orderDate,
    customerID,
    COUNT(*) OVER() Totalorder          (1 use case)
    COUNT(*) OVER(PARTITION BY customerID) ordersbycust.
FROM Sales.orders                       (2 use case)
```

#3 use case : (Data Quality check) Detecting number of NULLS by comparing to total number of rows.

```
SELECT
  *,
  COUNT(*) OVER() Total cust,
  COUNT(store) OVER() Totalstore,
  COUNT(country) OVER() Totalcountries
FROM Sales.Customer.
```

Data Quality Issue

- Duplicate data to inaccurate in analysis.

COUNT() can be used to identify duplicates.

```
SELECT
  orderID,
  COUNT(*) OVER(PARTITION BY orderID) checkPK
FROM Sales.order.
```

↓

SELECT Duplicate id (Greater than 1)

```
SELECT
  *
FROM (
  SELECT
    orderID,
    COUNT(*) OVER (partition by orderID)
    FROM Sales.orders Archive  checkPK
) t WHERE checkpk > 1
```

use case 4: (Identify Duplicates) :

Identify duplicate rows to improve data quality.

#SUM. : Returns the sum of values within a window.

#use case 1: * Overall analysis:- Quick summary or snapshot of entire dataset.

#use case 2: * Total per Groups :- Group-wise analysis, to understand patterns within different categories.

↓ SELECT
OrderID,
OrderDate,
ProductID,
Sales,
SUM(Sales) OVER() TotalSales,
SUM(Sales) OVER(Partition By ProductID) TotalProductSales
FROM Sales.Orders

use case 1 → TotalSales
use case 2 → TotalProductSales

• Part to whole analysis
→ Compare current sale to Total sales

• Compare to Extreme analysis
→ Compare current sales to Highest or lowest

• Compare to average analysis
→ Help to evaluate whether a value is above or below the average.

Ex:- percentage contribution (part to whole analysis)

```
SELECT
    OrderID,
    ProductID,
    Sales,
    SUM(Sales) OVER() TotalSales,
    ROUND(CAST(Sales AS FLOAT)/SUM(Sales) OVER()
        * 100, 2) PercentageOfTotal
FROM Sales.Orders
```

AVG(): Return average values within a window.

* If NULL in rows first remove NULL/ Handle NULL and then do avg()

Eg:
```
SELECT
    orderID,
    orderDate,
    Sales,
    AVG(Sales) OVER() AvgSales
    AVG(Sales) OVER(PARTITION BY ProductID) avgsales By products
```

#1 use case: overall analysis: Quick Summary of entire data.

#2 use case: Total per Groups: Group-wise analysis, to understand patterns within different categories

Eg:
```
SELECT
    CustomerID,
    LastName,
    Score,
    COALESCE(Score, 0),
    AVG(COALESCE(Score, 0)) OVER() AvgScoreWithNull
FROM Sales.Customers
```

* To Handle NULLs

Eg: Compare to average
```
SELECT *
FROM (
    SELECT
        OrderID,
        ProductID,
        Sales,
        AVG(Sales) OVER() AvgSales
    FROM Sales.Orders
) t
WHERE Sales > AvgSales
```

#3 use case:

Compare to average: Help to evaluate whether a value is above/below the average

#MIN & MAX : Returns highest value in window.

↓
Returns lowest value in the window.

Ex: SELECT
orderID,
productID,
Sales,
MAX(Sales) OVER() HighestSales
MIN(Sales) OVER() LowestSales
MAX(Sales) OVER(PARTITION BY ProductID) HighestSalesByProduct
MIN(Sales) OVER(PARTITION BY productID) LowestSalesByProduct
FROM Sales.Orders

#1 use case: Overall analysis: Quick summary of entire data.

#2 use case: Total per Groups: Group-wise analysis, to understand patterns within different Categories

Ex: Employee with highest salaries.

```
SELECT
*
FROM (
    SELECT
    *,
    MAX(Salary) OVER() HighestSalary
    FROM Sales.Employees
) t
WHERE Salary = HighestSalary
```

Ex: SELECT
orderID,
productID,
Sales,
MAX(Sales) OVER() HighestSales,
MIN(Sales) OVER() LowestSales,
Sales - MIN(Sales) OVER() DeviationFromMin
~~Sales - MAX(Sales) OVER()~~ DeviationFromMax
MAX(Sales) - Sales OVER() DeviationFromMax

#3 use case: Compare to extreme
Help to evaluate how well a value is performing relative to the extremes

Distance from Extreme

- Lower the deviation, the closer the data point is to the extreme.

Analytical use case

- Running & Rolling Total.
- Tracking : Tracking current sales with Target Sales.
- Trend analysis : providing Insights into Historical patterns.

* They aggregate sequence of members, and the aggregation is updated Each time a new member is added.

↳ Analysis over time

Running Total : Aggregate all values from beginning up to the current point without dropping off older data.

Rolling Total : aggregate all values within a fixed time window (eg 30 days) As new data is added, oldest data point will be dropped.

Rolling/Shifting window.

MOVING AVERAGE :

RUNNING AVERAGE

```
SELECT
    orderID,
    productID,
    OrderDate,
    Sales,
    AVG(Sales) OVER(PARTITION BY productID) avg_pro.
    AVG(Sales) OVER (PARTITION BY productID ORDER BY orderDate) Movingavg.
FROM Sales.orders
```

Q:- Calculate Moving avg of Sales per each product over Time

Q. Including only Next order.

↓

customize Frame : ROWS BETWEEN CURRENT ROW AND 1 FOLLOWING.

RANKING WINDOW FUNCTION

Task: Rank product based on their sales.

Note: SQL sort data as first step.

product	sales
E	70
B	30
A	20
C	10
D	5

product	sales	Integ-based Ranking
E	70	1
B	30	2
A	20	3
C	10	4
D	5	5

Discrete value

Find top 3

product	sales	percentage-based ranking
E	70	0
B	30	0,25
A	20	0,5
C	10	0,75
D	5	1

Continuous value.

Find 20%.

Contribution of data.

Top / Bottom N analysis

- ROW_NUMBER()
- DENSE_RANK()
- NTILE()
- RANK()

Distribution analysis

- CUME_DIST()
- PERCENT_RANK()

Syntax

```
RANK() OVER (PARTITION BY productID ORDER BY sales)
```

- RANK() — exp. must be empty
- PARTITION BY productID — optional
- ORDER BY sales — Required

ROW_NUMBER. Assign a unique number to Each row.

- It doesn't handle ties.

Two row sharing same value, they will not share same rank.

Sales	Rank
100	1
80	2
80	3
50	4
20	5

→ unique Ranking without gap / skipping.

ROW_NUMBER() OVER(ORDER BY Sales DESC)

→ Doesn't handle ties.

Ex:
```
SELECT
    OrderID,
    ProductID,
    Sales,
    ROW_NUMBER() OVER(ORDER BY Sales DESC) t
FROM Sales.Orders.
```

RANK.

- Assign a Rank to each row.
- It handle ties
  - It leaves gap in ranking.

Sales	Rank
100	1
80	2
80	2
50	4
20	5

→ Shared Ranking.
→ Leaving gaps (Ranking)
→ Handle ties
• ROW_NUMBER() OVER(ORDER BY Sales DESC) RANK

Ex:
```
SELECT
    OrderID,
    Sales,
    RANK() OVER(ORDER BY Sales DESC)
FROM Sales.Orders.
```

DENSE_RANK()

- Assign a ~~ROW~~ rank to each ROW.
- Handle ties.
- doesn't leave gaps in ranking.

Sales	Rank
100	1
80	2
80	2
50	3
20	4

- Shared Ranking
- Handle ties,
- Don't leave gap in Ranking.

DENSE_RANK() OVER(ORDER BY Sales DESC) t

Ex
```
SELECT
    *,
    DENSE_RANK() OVER(ORDER BY sales DESC) t
FROM order_sales.orders
```

TOP-IN-ANALYSIS

1) TOP highest sales per each product

```
( SELECT
    orderID,
    productID,
    Sales,
    ROW_NUMBER() OVER(PARTITION BY productID
    ORDER BY Sales DESC) Rankbyproduct
  FROM Sales.orders
) t  ⟶ subquery

  SELECT
    *
  FROM (
        t ⟶ subquery
      )
  WHERE Rankbyproduct = 1
```

use case: analyze top performer to do Targeted Marketing.

2) BOTTOM-IN-analysis.

lowest 2 customer based on their sales.

use case: Help analysis the underperformance to manage risks and to do optimizations

Rule: columns used in group by and window function must be the same

```
SELECT
  *
FROM (
    SELECT
      customerID,
      SUM(Sales) TotalSales,
      ROW_NUMBER() OVER(ORDER BY SUM(Sales)) RankCust
    FROM Sales.Orders ) t
WHERE RankCust <= 2
```

Generate unique IDs

- assign unique ID to row of 'ordersArchive' table.

#use case: Helps to assign a unique identifier for each row to help paginating

Ex:- SELECT
ROW_NUMBER() OVER(ORDER BY OrderID) uniqueID,
*
FROM Sales.OrdersArchive.

Paginating : process of breaking down a large data into smaller, more mangeable chunks.

Identify Duplicates.

#use case:- Identify and Remove Duplicates to improve data quality.

Ex: Identify duplicate row. return a clean result. SELECT
*
FROM (
SELECT
ROW_NUMBER() OVER(PARTITION BY OrderID ORDER BY CreationTime DESC) rn,
*
FROM Sales.OrdersArchive
) t
WHERE rn = 1 / TO collect only bad data.
WHERE rn > 1

NTILE Divides the rows into a specified number of approximately equal groups (buckets)

NTILE(2) OVER(ORDER BY Sales DESC)

Sales	NTILE
100	1
80	1
80	1
50	2
30	2

Bucketsize = No. of row / No. of bucket

2 = 5/2

SQL Rule
Larger group comes first.

Ex: 2
```
SELECT
    OrderID,
    Sales,
    NTILE(3) OVER(ORDER BY Sales DESC) 3 Bucket
FROM Sales.Orders
```

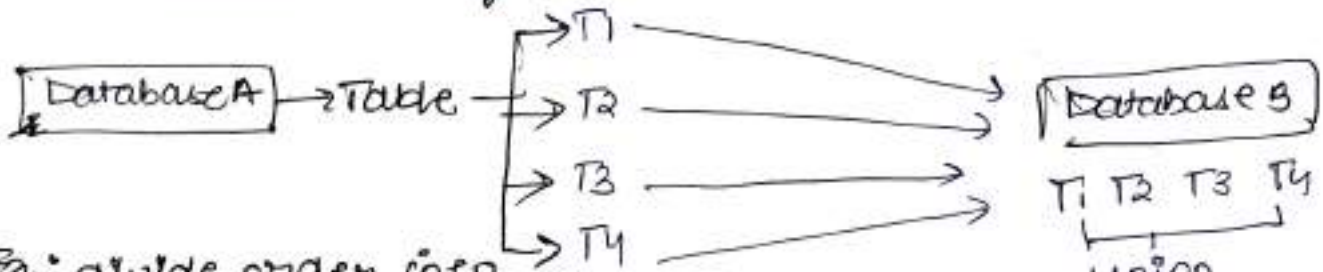
* NTILE USE CASE

Data segmentation : Divides the datasets into distinct subsets based on certain criteria.

Ex:- segment all order into 'High, Medium, Low'

```
SELECT
    *,
    CASE WHEN BUCKETS = 1 THEN 'High'
         WHEN Buckets = 2 THEN 'Medium'
         WHEN BUCKETS = 3 THEN 'Low'
    END series_segmentation
FROM ( SELECT
        OrderID,
        Sells,
        NTILE() OVER (ORDER BY Sales DESC)
            BUCKETS
       FROM Sales.Orders
)t
```

Load Balancing NTILE(4)



Ex: divide order into 2 group.

```
SELECT
    NTILE(2) OVER(ORDER BY OrderID) Buckets,
    *
FROM Sales.Orders
```

PERCENTAGE BASED RANKING

#CUME_DIST : Cumulative distribution calculates the distribution of data points within a window.

Sales	DIST
100	0.2
80	0.6
80	0.6
50	0.8
30	1

$$CUME\_DIST = \frac{\text{position No}}{\text{No. of Rows}}$$

$$= \frac{1}{5}$$

#Rule
position of last occurance of the same value

CUM_DIST() OVER(ORDER BY sales DESC)

PERCENT_RANK : calculate the relative position of each ROW.

Sales	Pos
100	0
80	0,25
80	0,25
50	0.75
30	1

$$percent\_rank = \frac{position\ no - 1}{No.\ of\ Row - 1}$$

$$= \frac{1-1}{5-1} = \frac{0}{4} \Bigg| = \frac{3}{4}$$

$$= \frac{2-1}{5-1} = \frac{1}{4} \Bigg|$$

Rule : position of first occurance of the same value.

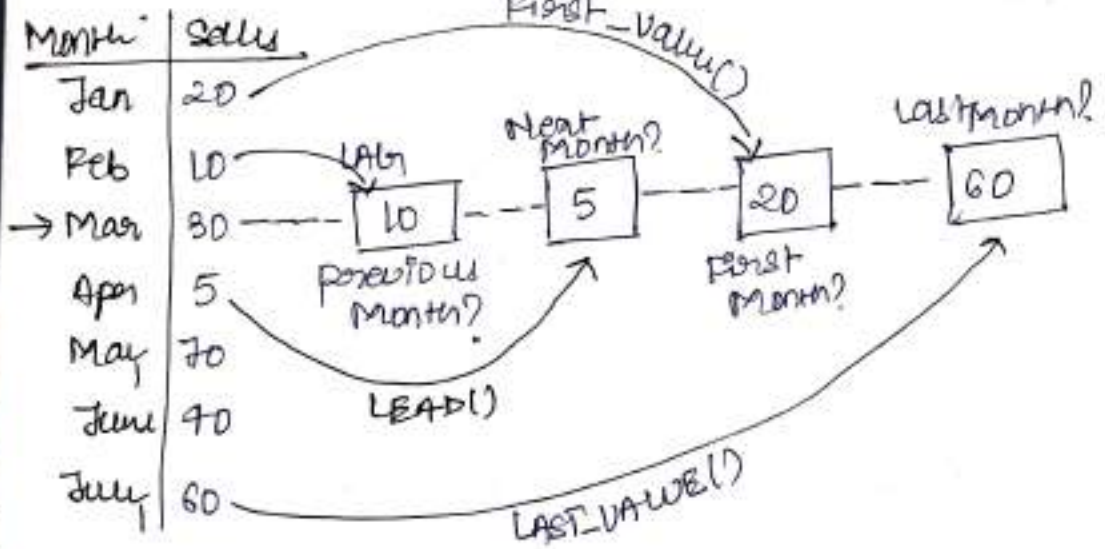
Inclusive
- current ROW is included.

Exclusive
- current ROW is excluded.

Ex- Find product that fall within the highest 40% of price.

```
SELECT
    *,
    CONCAT(DISRank * 100, '%') DistRankPerc
FROM (
        SELECT
            product,
            price,
            CUME_DIST() OVER(ORDER BY PRICE DESC)
                DISRank
            FROM sales.products
        ) t
WHERE DISRank <= 0.4
```

Value window analytical function.



Access value from other ROW.

MIN/MAX

LEAD() : Access value from NEXT ROW.

LAG() : Access value from previous ROW.

Syntax.

LEAD & LAG (syntax)

[PARTITION BY optional]

LEAD(sales, 2, 10) OVER(PARTITION BY productID ORDER BY orderdate)

(ORDER BY is required)

Expression is required (any data type)

offset (optional)

Number of rows forward or backward from current row.

Default = 1

default_value (optional)

Returns default value if next/previous row is not available.

Default = NULL

LEAD(Sales, 2, 0) OVER(ORDER BY MONTH)

Month	Sales	LEAD
Jan	20	30
Feb	10	5
Mar	30	0
Apr	5	0

LAG(sales, 2, 0) OVER(ORDER BY month)

Month	Sales	LAG
JAN	20	0
Feb	10	0
Mar	30	20
Apr	5	10

USE CASE

CMOM) MONTH-OVER-MONTH analysis.

Ex: MOM performance by finding % change in sales b/w current and previous month.

TIME SERIES ANALYSIS.

process to analyzing the data to understand pattern, trends and behaviours over time.

YEAR-OVER-YEAR (YOY)

Analyse the overall growth or decline the business performance over years/time.