W3 - WINDOW FUNCTIONS

WINDOW Functions allow us to perform calculations across a specific set of rows related to the current row. These calculations happen within a defined window of data, and they are particularly useful for **aggregates, ranking,** and **cumulative totals** without altering the dataset.

- RANK() - skips after ties (i.e 1,2,3,3,3,6,7,..)
- ROW_NUMBER() - returns the row number when data ordered according to ORDER BY statement
- DENSE_RANK() - doesn't skip (i.e 1,2,3,3,3,4,5,..)
- ORDER BY

The OVER clause is KEY to defining this window. It partitions the data into different SETS (using PARTITION BY clause) and orders them (using the ORDER BY clause). These windows enable functions like:

- SUM()
- AVG()
- ROW_NUMBER()
- RANK()
- DENSE_RANK()

{FUNCTION()} OVER (PARTITION BY {column}) AS ...

```
SELECT OrderID, EmployeeID
, COUNT(*) OVER (PARTITION BY EmployeeID) AS 'Employee Order Count'
FROM Orders
ORDER BY OrderID
```

- Using COUNT alongside PARTITION BY
- {FUNCTION} OVER (PARTITION BY {column}) AS . . .
- Calculates COUNT(*) separately for each EmployeeID

🧠Aggregate Window Functions

These compute aggregates over a defined window:

- `SUM()`
- `AVG()`

- `COUNT()`
- `MIN()`
- `MAX()`

🧠 Example:

SUM(sales) `OVER (PARTITION BY region ORDER BY date)`

🆙 Ranking Functions

Used to assign a rank or row number:

- `ROW_NUMBER()` – Unique row number per partition
- `RANK()` – Gives same rank for ties, but skips next rank
- `DENSE_RANK()` – Same as RANK but no rank gaps
- `NTILE(n)` – Divides rows into `n` equal-sized buckets

🆙 Example:

RANK() `OVER (PARTITION BY department ORDER BY salary DESC)`

3. Value Functions (Offset & Navigation)

Allow access to other rows:

- `LAG()` – Value from a previous row => LAG(return_value,offset,default)
- `LEAD()` – Value from a following row => LEAD(return_value,offset,default)
- `FIRST_VALUE()` – First value in window
- `LAST_VALUE()` – Last value in window

⚠️ The catch: default window frames

By default, SQL window functions like `LAST_VALUE()` operate over a **moving frame** — typically:

ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

- `NTH_VALUE()` – nth value in window

🧠 Example:

```
LAG(salary, 1) OVER (PARTITION BY department ORDER BY hire_date)
```

{ROW_NUMBER()} OVER (PARTITION BY {column}) AS .

```
SELECT ProductName
, UnitPrice
, ROW_NUMBER() OVER (ORDER BY UnitPrice DESC) AS 'Row Number'
FROM Products
```

- ROW_NUMBER() returns the row number when data ordered according to ORDER BY statement
- ORDER BY can be ASCENDING (ASC, default) or DESCENDING (DESC)
- Identical values ordered arbitrarily
  - RANK() and DENSE_RANK() will assign same number for ties
  - RANK() skips after ties (e.g. 1, 2, 3, 3, 3, 6, 7…)
  - DENSE_RANK() doesn't skip (e.g. 1, 2, 3, 3, 3, 4, 5…)
- Independent to the ORDER BY clause in the main query

```
SELECT ProductName, UnitPrice
, LEAD(ProductName, 1, NULL) OVER (ORDER BY UnitPrice)
    AS 'Next Product Price'
FROM Products
```

- LEAD( ) looks ahead a certain number of rows
- Takes 3 arguments:
  - Column to Return
  - Number of Rows to Look Forward
  - Default Value if Nothing Can Be Returned
- LAG( ) works in same way, but looks behind

```
SELECT ProductName, SupplierID
, FIRST_VALUE(UnitsOnOrder) OVER (PARTITION BY SupplierID ORDER BY
UnitsOnOrder DESC)
    AS 'Most Units on Order from Supplier'
FROM Products
```

- FIRST_VALUE( ) will return the first row based on the ORDER BY, for each partition
- It takes one argument – column name to return from the first row

```
LAST_VALUE(OrderID) OVER (PARTITION BY EmployeeID ORDER BY OrderID
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
```

- Along with the PARTITION BY and ORDER BY clauses, there is also the ROW or RANGE clause
- The default (above) means run the analytic function on all rows between UNBOUNDED PRECEDING (the first row of a partition) and the current row.
- So by default, LAST_VALUE( ) only looks between the start of the partition and the current row, so it returns the current row!
- We need to specify the window to return the last value for the partition.

```
{ ROWS / RANGE }

BETWEEN

[ { UNBOUNDED / N } PRECEDING
  CURRENT ROW ]

AND

[ { UNBOUNDED / N } FOLLOWING
  CURRENT ROW ]
```

```
SELECT OrderID, EmployeeID
, LAST_VALUE(OrderID) OVER (PARTITION BY EmployeeID ORDER BY OrderID
    ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
    AS 'Last Order for Employee'
FROM Orders
```

- Changing the default window clause
- At each row, the analytic function now considers the entire partition before returning the Last Value

```
SELECT OrderID, OrderDate, Freight
, AVG(Freight) OVER (ORDER BY OrderDate
    ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
    AS '3pt Moving Avg'
FROM Orders
```

- By specifying the number of preceding and following rows, we can calculate a moving average