15. Window Functions — Deep Dive

Window functions let you look across rows *without* collapsing them. They're like telescopes you mount on your SELECT clause: you see the neighborhood around each row while still getting one output row per input row.

A window function = `function() OVER (PARTITION BY ... ORDER BY ... <frame clause>)`

Why window functions matter

They solve problems that are painful with subqueries or joins:

- running totals
- percentiles
- rankings and row numbering
- comparing a row to the previous or next row
- computing moving averages
- detecting gaps
- advanced analytics (distribution functions)

They're indispensable in data engineering pipelines when shaping fact tables, deduplicating changefeeds, or building analytic summaries.

The OVER() clause — the control center

`OVER()` defines the *window* of rows a function can see.

- `PARTITION BY` divides rows into groups.
- `ORDER BY` defines the sequence within each partition.
- `ROWS` or `RANGE` defines the frame of rows accessible.

Example structure:

```
SUM(SalesAmount) OVER (
    PARTITION BY CustomerID
    ORDER BY OrderDate
    ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
) AS RunningTotal
```

ROW_NUMBER(), RANK(), DENSE_RANK()

ROW_NUMBER()

Assigns a unique sequence per partition.

```
SELECT
  OrderID,
  ROW_NUMBER() OVER (PARTITION BY CustomerID ORDER BY OrderDate) AS Seq
FROM Sales.Orders;
```

Great for deduplication patterns:

```
WITH Ranked AS (
  SELECT *, ROW_NUMBER() OVER (PARTITION BY CustomerID ORDER BY UpdatedAt DESC) AS rn
  FROM Staging.Customers
)
SELECT * FROM Ranked WHERE rn = 1;
```

RANK() and DENSE_RANK()

```
RANK():       1,2,2,4
DENSE_RANK(): 1,2,2,3
```

LAG() and LEAD() — seeing previous and next rows

Compare a row to its neighbors.

```sql
SELECT
  OrderDate,
  Revenue,
  LAG(Revenue)  OVER (ORDER BY OrderDate) AS PrevRev,
  LEAD(Revenue) OVER (ORDER BY OrderDate) AS NextRev
FROM Finance.DailyRevenue;
```

Useful for detecting shifts, gaps, and change deltas:

```sql
SELECT
  OrderID,
  Quantity - LAG(Quantity) OVER (ORDER BY OrderID) AS DiffFromPrev
FROM Inventory.Movements;
```

FIRST_VALUE(), LAST_VALUE(), NTH_VALUE()

FIRST_VALUE

```sql
FIRST_VALUE(Price) OVER (ORDER BY Price ASC)
```

LAST_VALUE

Needs a frame fix; default frames can make it return the current row.

```sql
LAST_VALUE(Price) OVER (
  ORDER BY Price
  ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
)
```

Aggregate windows — the running total classics

Running totals

```sql
SUM(Amount) OVER (
  PARTITION BY AccountID
  ORDER BY TxnDate
  ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
) AS Balance
```

Moving averages

```sql
AVG(Temperature) OVER (
  ORDER BY ReadingTime
  ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
) AS SevenPointAvg
```

ROWS vs RANGE — subtle but important

ROWS

Counts physical rows.

```sql
ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
```

Always deterministic.

RANGE

Groups rows by ORDER BY value equivalence.

```sql
RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
```

Not allowed for non-numeric ORDER BY expressions.

NTILE() — bucketing

Create percentiles or quartiles.

```
SELECT
  CustomerID,
  NTILE(4) OVER (ORDER BY TotalSpent) AS Quartile
FROM Analytics.CustomerSpend;
```

Perfect for segmentation.

Distribution functions: PERCENT_RANK(), CUME_DIST()

PERCENT_RANK

Position of a row in its partition between 0 and 1.

```
PERCENT_RANK() OVER (ORDER BY Score)
```

CUME_DIST

Cumulative distribution; fraction of rows ≤ current row.

```
CUME_DIST() OVER (ORDER BY Score)
```

Practical data engineering patterns

1. Deduplicating CDC streams (pick latest version)

```
WITH v AS (
  SELECT *, ROW_NUMBER() OVER (
            PARTITION BY BusinessKey ORDER BY UpdatedAt DESC) AS rn
  FROM Raw.CDC_Events
)
SELECT * FROM v WHERE rn = 1;
```

2. Detecting gaps in sequences

```sql
SELECT
  EventID,
  EventDate,
  LAG(EventDate) OVER (ORDER BY EventDate) AS Prev,
  DATEDIFF(day, Prev, EventDate) AS GapDays
FROM Logs.Events;
```

3. Building SCD Type 2 boundaries

```sql
SELECT
  EmployeeID,
  StartDate,
  LEAD(StartDate) OVER (PARTITION BY EmployeeID ORDER BY StartDate) AS EndDate
FROM HR.EmployeeHistory;
```

Produces inferred end dates.

4. Running inventory balances

```sql
SELECT
  ItemID,
  TxnDate,
  SUM(DeltaQty) OVER (
      PARTITION BY ItemID ORDER BY TxnDate
      ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
  ) AS StockLevel
FROM Warehouse.InventoryMoves;
```

Final notes

Window functions don’t reduce row count. They augment each row with context. They execute *after* WHERE but *before* ORDER BY in the logical query order.

This chapter completes the analytic machinery of T-SQL. Next up, if you continue the series, we can explore indexing, transactions, performance tuning, or error handling — wherever your data engineering map leads.