Mastering Window Functions in MS SQL (Beginner → Advanced)

Window functions behave like tiny mathematicians sitting on your query results, peeking at neighboring rows without collapsing everything into a single aggregated row. They let you compute running totals, rankings, moving averages, percentiles, and other analytic calculations with elegance.

This guide walks from fundamentals to advanced mastery using clear explanations and SQL Server–friendly patterns.

---

1. What Is a Window Function?

A window function performs a calculation across a set of rows (the *window*) that are related to the current row. Unlike GROUP BY, window functions **do not collapse rows**. They add insight *alongside* existing data.

The canonical structure:

```
<window_function>() OVER (
    [PARTITION BY ...]
    [ORDER BY ...]
    [ROWS/RANGE ...]
)
```

Think of PARTITION BY as slicing the table into segments, ORDER BY as defining the timeline or order inside each slice, and ROWS/RANGE as defining the window frame—how many neighboring rows the function can observe.

---

2. Basic Window Functions

2.1 ROW_NUMBER

Assigns sequential row numbers.

```sql
SELECT
    SalesOrderID,
    CustomerID,
    ROW_NUMBER() OVER (ORDER BY SalesOrderID) AS OrderPosition
FROM Sales.SalesOrderHeader;
```

Helps extract “nth” row in a group or deduplicate data.

2.2 RANK and DENSE_RANK

RANK leaves gaps; DENSE_RANK does not.

```sql
SELECT
    ProductID,
    ListPrice,
    RANK() OVER (ORDER BY ListPrice DESC) AS PriceRank,
    DENSE_RANK() OVER (ORDER BY ListPrice DESC) AS DensePriceRank
FROM Production.Product;
```

Useful when determining top-selling products or price tiers.

2.3 NTILE

Splits rows into N equal buckets.

```sql
SELECT
    ProductID,
    ListPrice,
    NTILE(4) OVER (ORDER BY ListPrice) AS PriceQuartile
FROM Production.Product;
```

Essential for quantile-based segmentation.

---

3. Aggregate Window Functions

They behave like GROUP BY aggregates, but without collapsing rows.

3.1 SUM OVER

```sql
SELECT
    SalesOrderID,
    CustomerID,
    TotalDue,
    SUM(TotalDue) OVER (PARTITION BY CustomerID) AS CustomerTotal
FROM Sales.SalesOrderHeader;
```

Calculates total amount spent by each customer.

3.2 Running Total

```sql
SELECT
    SalesOrderID,
    OrderDate,
    TotalDue,
    SUM(TotalDue) OVER (
        ORDER BY OrderDate
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) AS RunningSales
FROM Sales.SalesOrderHeader;
```

Crucial for time-series analytics.

3.3 Moving Average

```sql
SELECT
    SalesOrderID,
    OrderDate,
    TotalDue,
    AVG(TotalDue) OVER (
        ORDER BY OrderDate
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
    ) AS MovingAvg3
FROM Sales.SalesOrderHeader;
```

Smooths fluctuations in sales performance.

---

4. Value Window Functions

4.1 LAG and LEAD

Peek at previous or next row.

```
SELECT
    SalesOrderID,
    OrderDate,
    TotalDue,
    LAG(TotalDue, 1) OVER (ORDER BY OrderDate) AS PreviousOrderAmount,
    LEAD(TotalDue, 1) OVER (ORDER BY OrderDate) AS NextOrderAmount
FROM Sales.SalesOrderHeader;
```

Ideal for difference calculations.

4.2 FIRST_VALUE and LAST_VALUE

```
SELECT
    SalesOrderID,
    OrderDate,
    TotalDue,
    FIRST_VALUE(TotalDue) OVER (ORDER BY OrderDate) AS FirstOrderAmount,
    LAST_VALUE(TotalDue) OVER (
        ORDER BY OrderDate
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) AS LastOrderAmount
FROM Sales.SalesOrderHeader;
```

Note: LAST_VALUE needs a window frame override.

---

5. Understanding Window Frames

Frames determine which rows "count" for the function.

5.1 ROWS vs RANGE

- **ROWS** counts physical rows.
- **RANGE** groups by ORDER BY values.

Example of ROWS:

```
ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
```

Example of RANGE:

```
RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
```

Frames allow precise control for analytics.

---

6. Advanced Patterns

6.1 Gaps-and-Islands (Identifying continuous sequences)

```
WITH x AS (
    SELECT
        SalesOrderID,
        CustomerID,
        ROW_NUMBER() OVER (ORDER BY SalesOrderID) AS rn
    FROM Sales.SalesOrderHeader
)
SELECT
    *,
    SalesOrderID - rn AS IslandGroup
FROM x;
```

This trick identifies consecutive sequences elegantly.

6.2 Percentiles

```
SELECT
    ProductID,
    ListPrice,
    PERCENT_RANK() OVER (ORDER BY ListPrice) AS PricePercentile
FROM Production.Product;
```

Great for statistical distributions.

6.3 Top 1 Row per Group (without subqueries)

```sql
SELECT *
FROM (
    SELECT
        ProductID,
        ListPrice,
        ROW_NUMBER() OVER (PARTITION BY ProductSubcategoryID ORDER BY ListPrice DESC)
AS rn
    FROM Production.Product
) q
WHERE rn = 1;
```

Faster and cleaner than correlated subqueries.

---

7. Performance Considerations

1. ORDER BY inside window functions can be expensive on large tables.
2. Proper indexing on partition + order keys dramatically improves speed.
3. Mixing many window functions in the same query is fine; SQL Server may reuse the sort.
4. Avoid complicated RANGE frames unless necessary—they are slower.

---

8. Real Data Engineering Scenarios

8.1 Customer Lifetime Value (CLTV)

```sql
SELECT
    CustomerID,
    OrderDate,
    TotalDue,
    SUM(TotalDue) OVER (PARTITION BY CustomerID) AS LifetimeSpend,
    ROW_NUMBER() OVER (PARTITION BY CustomerID ORDER BY OrderDate) AS PurchaseNumber
FROM Sales.SalesOrderHeader;
```

Adds customer stage analytics.

8.2 Detecting Anomalous Spikes in Sales

```sql
SELECT
    OrderDate,
    TotalDue,
    AVG(TotalDue) OVER (
        ORDER BY OrderDate
        ROWS BETWEEN 7 PRECEDING AND CURRENT ROW
    ) AS RollingWeekAvg,
    TotalDue - LAG(TotalDue) OVER (ORDER BY OrderDate) AS DayOverDayChange
FROM Sales.SalesOrderHeader;
```

Helps with anomaly detection pipelines.

8.3 Vendor Performance Trend

```sql
SELECT
    VendorID,
    PurchaseOrderID,
    OrderDate,
    SUM(TotalDue) OVER (PARTITION BY VendorID ORDER BY OrderDate) AS
VendorCumulativeSpend
FROM Purchasing.PurchaseOrderHeader;
```

Useful in procurement analytics.

---

9. Summary

Window functions unlock a powerful analytic layer inside SQL itself. They let you treat your result set as a dynamic playground—without temp tables, without subqueries, and without losing detail.

Next steps often include practicing with realistic business questions, building combined patterns (like moving averages over partitions), and mixing window analytics with CTE-based transformations to create full data engineering logic.