4. T-SQL SELECT — Anatomy and Examples

This section builds on your relational foundations and shows how a `SELECT` statement actually *thinks*. SQL has a logical processing order that feels a bit like reading a book backward, but once you internalize it, queries become far easier to reason about.

The goal here is to understand the mental model behind queries and then use that model to write clean, predictable SQL.

4.1 The Logical Order of a SELECT Query

A SQL query **does not** execute in the order it is written. It follows a logical order that looks like this:

1. `FROM` (and joins)
2. `WHERE` (row filtering)
3. `GROUP BY` (bucket rows into groups)
4. `HAVING` (filter groups)
5. `SELECT` (projection — choose what to output)
6. `ORDER BY` (sort result)
7. `OFFSET/FETCH` (pagination)

The written order is just the surface grammar; the logical order is the SQL engine's actual reasoning path.

4.2 Simple Example to See the Order

```sql
SELECT FirstName, LastName
FROM dbo.Customers
WHERE CreatedAt >= '2024-01-01'
ORDER BY LastName;
```

Here's how SQL processes it:

- Start with `FROM dbo.Customers`
- Apply the filter: keep rows where `CreatedAt` is on or after 2024-01-01
- Project only the columns `FirstName`, `LastName`
- Sort the final result by `LastName`

4.3 SELECT Clause Anatomy

The key idea: **SELECT happens after GROUP BY logically**, so every column in the SELECT must be either:

- part of the grouping columns, or
- an aggregated expression (`SUM()` , `COUNT()` , etc.)

Example — valid:

```
SELECT CustomerID, SUM(Amount) AS TotalSpent
FROM dbo.Payments
GROUP BY CustomerID;
```

Example — invalid:

```
SELECT CustomerID, Amount
FROM dbo.Payments
GROUP BY CustomerID;  -- ❌ Amount is not aggregated
```

4.4 The FROM Clause — the query's "starting universe"

Everything begins with the tables and joins declared in the FROM.

Example:

```
FROM dbo.Customers c
JOIN dbo.Orders o ON o.CustomerID = c.CustomerID
JOIN dbo.OrderItems oi ON oi.OrderID = o.OrderID
```

This produces a combined rowset of all three tables before any filtering or grouping is applied.

4.5 WHERE Clause — filtering rows early

Applied immediately after the FROM. This determines **which rows survive** into the next steps.

```
WHERE o.OrderDate >= '2024-01-01'
```

The WHERE clause can use:

- comparisons (`=` , `<>` , `>` , `<`)
- ranges (`BETWEEN`)
- `IN` , `EXISTS`

- wildcard patterns (`LIKE 'A%'`)
- boolean logic (`AND` , `OR`)

Important: **WHERE cannot use aggregates.** Use HAVING for that.

4.6 GROUP BY — creating buckets of rows

Once rows are filtered, SQL groups them.

Example:

```
GROUP BY c.CustomerID, c.Name
```

Each unique combination becomes a group. You can then apply aggregates:

- `SUM()`
- `AVG()`
- `COUNT()`
- `MAX()`
- `MIN()`

4.7 HAVING — filtering groups (not rows)

HAVING comes after GROUP BY and filters **aggregated** results.

Example:

```
HAVING SUM(oi.Quantity * oi.UnitPrice) > 1000
```

This removes groups (customers) whose total spend is under 1000.

4.8 ORDER BY — giving meaning to order

SQL doesn't guarantee row order unless you use ORDER BY.

```
ORDER BY TotalSpent DESC;
```

You can use:

- column names
- aliases
- expressions

You can also use ordinal positions (like `ORDER BY 3`), but this is discouraged because it makes queries fragile.

4.9 Example — Putting It All Together

```
SELECT
    c.CustomerID,
    c.Name,
    SUM(oi.Quantity * oi.UnitPrice) AS TotalSpent
FROM dbo.Customers c
JOIN dbo.Orders o ON o.CustomerID = c.CustomerID
JOIN dbo.OrderItems oi ON oi.OrderID = o.OrderID
WHERE o.OrderDate >= '2024-01-01'
GROUP BY c.CustomerID, c.Name
HAVING SUM(oi.Quantity * oi.UnitPrice) > 1000
ORDER BY TotalSpent DESC;
```

This query answers a business question: “Which customers spent more than 1000 after January 1st, 2024?”

4.10 Practical Tips

- Avoid `SELECT *` in production.
- Use table aliases (`c`, `o`, `oi`) for readability.
- Always specify `schema.table` — e.g., `dbo.Customers`.
- Use `ORDER BY` whenever deterministic ordering is required.
- Prefer SARGable predicates (no functions on indexed columns).

4.11 Mini Exercise

Use the tables created in the previous section:

Write a query that returns each product with the **total revenue generated** from all orders.

Hint:

- join `Products → OrderItems`
- revenue = `Quantity * UnitPrice`
- group by product

Try writing it first, then run it and explore the execution plan.