6. NULLs and Three-Valued Logic

NULLs sit at the philosophical edge of SQL Server. They don't mean zero, and they don't mean empty—they mean *unknown*. This single idea bends the entire logic engine in curious ways.

A NULL isn't a value you can compare. It's more like a shrug baked into the data: "I genuinely don't know what goes here." To accommodate such ambiguity, SQL Server uses something called **three-valued logic**.

The Nature of NULL

SQL Server treats NULL as missing information.

If a phone number is NULL, you aren't claiming the customer has no phone—you're saying you don't know it. This subtle difference has big consequences.

A comparison like `NULL = NULL` doesn't return true. Since both are unknown, SQL Server refuses to take a stance.

Enter Three-Valued Logic

Traditional logic has *true* and *false*. SQL Server adds a third: *unknown*.

Whenever a NULL participates in a comparison or logical expression, the result becomes *unknown* unless proven otherwise.

Imagine a tiny truth table for a simple comparison like `Age > 30`.

If `Age = 40` → true. If `Age = 20` → false. If `Age = NULL` → unknown. SQL Server can't decide.

This leads to interesting behavior. A WHERE clause filters out any row whose condition evaluates to false *or* unknown. Unknown doesn't get a pass.

Common Surprises

NULL equals nothing

`NULL = anything` is unknown.

`NULL <> anything` is also unknown.

The same goes for greater than, less than, or most comparisons.

WHERE filters out NULLs without asking

```
SELECT * FROM Sales WHERE Discount > 0;
```

If `Discount` is NULL for a row, SQL Server doesn't know whether it's greater than zero. It quietly drops the row.

NULLs inside aggregates behave oddly

`COUNT(column)` ignores NULLs.

`SUM(column)` ignores NULLs.

`AVG(column)` ignores NULLs.

It's often intentional, but if you expect zeros to stand in, you must handle them explicitly.

Tools for Taming NULL

SQL Server gives you a handful of functions so you stop tripping over unknowns.

`IS NULL` and `IS NOT NULL` let you check for NULL precisely.

`ISNULL(value, fallback)` replaces NULLs with another value, usually for downstream formatting.

`COALESCE(a, b, c...)` returns the first non-NULL expression from its arguments. It's ANSI-standard and works nicely in expressions.

`NULLIF(a, b)` returns NULL when two values match. Strange, poetic, and surprisingly useful.

Practical Examples

Example 1: Categorizing NULL values

```
SELECT
    ProductID,
    ISNULL(Discount, 0) AS AppliedDiscount
FROM Sales.OrderDetail;
```

This turns unknown discounts into honest zeroes.

Example 2: COALESCE for fallback logic

```
SELECT
    EmployeeID,
    COALESCE(WorkEmail, PersonalEmail, 'no-email@domain') AS ContactEmail
FROM HR.Employee;
```

It treats email fields like a ladder—SQL climbs from one to the next until it finds something non-NULL.

Example 3: Filtering correctly

```
SELECT *
FROM Inventory
WHERE QuantityOnHand IS NULL;
```

This catches genuinely missing quantities, not zero stock.

Real-World Impacts

ETL pipelines need careful handling of NULLs because data from external sources often arrives half-formed. A NULL in a timestamp column might mean the record hasn't been processed yet. Or it might signal corruption.

Dashboards and reports can mislead if they don't treat NULLs with intent. Dropped rows might hide missing data.

Joins involving NULLs can be tricky. Inner joins shrug and move on; outer joins preserve NULLs and expose what's missing.

Three-valued logic sometimes causes surprise, but once you internalize it, SQL queries feel less like brittle machinery and more like carefully engineered thought experiments.

This chapter completes the conceptual foundation for working with SQL truth. The next step is constraints—how SQL Server enforces structure, correctness, and discipline on your data.