**Failure #1: Not knowing what NULL means**

When referring to **NULL** within our data sets, we often use phrases such as “the **NULL** value” or “a value of **NULL**.” I do it all the time. I see it done all the time. Such phrases are so common that we think little of their use. But inherent in the phrase is the notion of **NULL** as an actual value. It is not. **NULL** is a non-value, a nonexistent value. It is not zero. It is not an empty string. A value cannot equal **NULL**. No two **NULL** values are equal.

A **NULL** value is often defined as one that is unknown or not applicable, but even these definitions can be open to debate. For example, a record might not include a customer’s birthdate because the salesperson didn’t ask or because the customer would not provide it, but the customer still knows that date, so it is hardly unknown, nor is it any less applicable. There can be many reasons for missing data, and there has been much debate about a precise meaning of **NULL**. If you need to attach a specific meaning to **NULL**, then *missing*or *absent*data is probably your safest bet.

Perhaps a better way to think of **NULL** is as a setting or marker that indicates if a data value does not exist. That certainly seems to be how SQL Server treats **NULL**. The database engine uses a special bitmap to track which columns in a row are **NULL** and which are not. The bitmap contains a bit for each column, with the bit set to 1 if the column is **NULL**, that is, if the value is missing.

**Failure #2: Treating NULL like a real value in comparisons**

Developers who don’t understand how **NULL** works will sometimes use comparison operators to compare an expression to a **NULL** value, resulting in a statement that looks similar to the following:

|  |  |
| --- | --- |
|  | SELECT Title, FirstName, MiddleName, LastName  FROM Person.Person  WHERE Title = NULL AND BusinessEntityID < 7; |

The assumption here is that **NULL** is a valid value and therefore can be compared to other **NULL** values. That means, in this case, the statement should return each row whose **Title** value is **NULL**. A quick check into the source data will show four rows that match the statement’s assumed logic. However, because the database engine cannot compare nothing to nothing, it will not return these rows and will instead provide us with an empty result set.

**Failure #3: Not knowing how ANSI\_NULL works**

In the last failure, I demonstrated why you should not use comparison operators against **NULL** values. Even if there are results to be had, a statement such as the following will likely not return them:

|  |  |
| --- | --- |
|  | SELECT Title, FirstName, MiddleName, LastName  FROM Person.Person  WHERE Title = NULL AND BusinessEntityID < 7; |

The word *likely* is key here. It turns out that the behavior I described is the default behavior, but not the only behavior. SQL Server supports a feature called **ANSI\_NULLS**, which determines how the database engine handles **NULL** comparisons. The option is a carry-over from the ghost of SQL Server Past and has been deprecated since 2005. But the setting still exists, it still works, and it can still cause all sorts of trouble.

By default, SQL Server installs with the **ANSI\_NULLS** option turned on, which gives us the behavior I’ve described so far. If we compare values to **NULL**, the database engine does not return those rows, even if **NULL** exists.

**Failure #4: Treating ISNULL the same as IS NULL**

The **ISNULL** function tends to cause a bit of confusion when first encountered, in part because of its name, which implies a meaning similar to the **IS** **NULL** operator. Another reason for confusion arises from the fact that products such as Microsoft Access and MySQL also include the **ISNULL** function, where it behaves much differently, more like the name suggests. But we’re talking SQL Server and need to understand how **ISNULL** is implemented here. A lack of understanding can lead to misuse and failure to take advantage of its real purpose in SQL Server.

Let’s looks at a few examples to demonstrate, starting with a simple query that returns products whose **Color** value is **NULL**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | SELECT Name AS ProductName, Color  FROM Production.Product  WHERE ProductID BETWEEN 940 AND 949    AND Color IS NULL; | | | |
| **ProductName** | **Color** |
| LL Touring Handlebars | NULL |
| HL Touring Handlebars | NULL |

If we’re not familiar with **ISNULL** in SQL Server or are transitioning in from Access, we might attempt to retrieve the same results by using **ISNULL** in our **WHERE** clause:

|  |  |
| --- | --- |
|  | SELECT Name AS ProductName, Color  FROM Production.Product  WHERE ProductID BETWEEN 940 AND 949    AND ISNULL(Color); |

Fortunately, this construction will generate the following error, which we’ll likely catch immediately, or at least during the Q&A phase:

|  |  |
| --- | --- |
|  | Msg 174, Level 15, State 1, Line 305  The isnull function requires 2 argument(s). |

Those of us with a background in MySQL might instead take a different approach with the **ISNULL** function by comparing it to 1 or 0, the function’s two possible return values:

|  |  |
| --- | --- |
|  | SELECT Name AS ProductName, Color  FROM Production.Product  WHERE ProductID BETWEEN 940 AND 949    AND ISNULL(Color) = 1); |

Once again, we receive an error message:

|  |  |
| --- | --- |
|  | Msg 174, Level 15, State 1, Line 322  The isnull function requires 2 argument(s). near ';'. |

By now, we should have checked the documentation to verify what this function is supposed to do and have come to the conclusion that, based on the syntax *and* the fairly explicit error messages, we should be adding a second argument, which we do:

|  |  |
| --- | --- |
|  | SELECT Name AS ProductName, Color  FROM Production.Product  WHERE ProductID BETWEEN 940 AND 949    AND ISNULL(Color, 'neutral'); |

Unfortunately, all we’ve done is make our predicate completely illogical, resulting in yet another (but different) error message:

|  |  |
| --- | --- |
|  | Msg 4145, Level 15, State 1, Line 339  An expression of non-boolean type specified in a context where a condition is expected, near ';'. |

The point of that this is that, in SQL Server, the purpose of the **ISNULL** function is simply to replace an expression’s value with a real value, if the original value is **NULL** (nonexistent). If we wanted to make the preceding example right, we would need to define our Boolean condition correctly:

|  |  |
| --- | --- |
|  | SELECT Name AS ProductName, Color  FROM Production.Product  WHERE ProductID BETWEEN 940 AND 949    AND ISNULL(Color, 'neutral') IN ('black', 'neutral'); |

Now the **SELECT** statement returns the results we want, rather than an error:That said, we can achieve the same thing by using **IS** **NULL**, without incurring the extra processing:

|  |  |
| --- | --- |
|  | SELECT Name AS ProductName, Color  FROM Production.Product  WHERE ProductID BETWEEN 940 AND 949    AND (Color = 'black' OR Color IS NULL); |

More often than not, the place we’ll want to use the **ISNULL** function is within our select list to replace the **NULL** values with real values:

|  |  |
| --- | --- |
|  | SELECT Name AS ProductName, ISNULL(Color, 'neutral')  FROM Production.Product  WHERE ProductID BETWEEN 940 AND 949; |

Be aware that, when using **ISNULL**, the database engine converts the replacement value to the data type of the original value, making its possible for data to get truncated or for the database engine to generate a conversion error. Make sure you read up on **ISNULL** before adding it to your code.

# Failure #5: Treating ISNULL the same as COALESCE

In the last failure, I tried to address any confusion there might be around the **ISNULL** function, but there’s another area of possible confusion, and that is in understanding the differences between **ISNULL** and **COALESCE**. Both functions let us replace **NULL** with a real value, but there are differences between the two that can be important to understand.

First off, the **ISNULL** function implemented in SQL Server is specific to that brand of T-SQL, whereas **COALESCE** is standard ANSI, making it more universal and, consequently, more portable, an important consideration if the possibility exists that you might one day want to point your apps to a different database system.

The **COALESCE** function also lets us include more than two expressions, as shown in the following **SELECT**statement:

The **COALESCE** function can actually be thought of as syntactical shorthand for a **CASE** expression:

|  |  |
| --- | --- |
|  | SELECT Name AS ProductName, Class, Style, ProductLine,    CASE      WHEN Class IS NOT NULL THEN Class      WHEN Style IS NOT NULL THEN Style      WHEN ProductLine IS NOT NULL THEN ProductLine      ELSE 'n/a'    END AS FirstNotNull  FROM Production.Product  WHERE ProductID IN(679, 706, 709, 711); |

The **SELECT** statement achieves the same results as the preceding example, but its construction is more complex. The main advantage of using a **CASE** expression is that it makes the logic clearer, for those who might need a little extra help.

Compared to the **COALESCE** function, the **ISNULL** function is much more limited:

|  |  |
| --- | --- |
|  | SELECT Name AS ProductName, Class, Style,    ISNULL(Class, Style) AS FirstNotNull  FROM Production.Product  WHERE ProductID IN(679, 706, 709, 711); |

In this case, all we’re saying is that if **Class** is **NULL**, then use **Style**, giving us the following results

As great as the **NULLIF** function is for handling certain situations, such as avoiding divide-by-zero errors, having **NULL** show up in our numeric calculations is often less than optimal. It’s not that SQL Server does anything surprising, but rather that you need to know what’s coming should you run into issues with your calculations.

# Failure #7: Assuming that 2 + NULL = 2

This all goes back to the idea that **NULL** is a non-value. It is nothing. It merely indicates that any real data value is missing. Consequently, when we try to add a value to nothing or subtract from nothing or multiply by nothing or divide by nothing or do anything by nothing, we end up with nothing, or **NULL**.

**Failure #8: Aggregating data without taking NULL into account**

As with numeric calculations, we must be prepared to handle aggregations that could include **NULL** values. However, unlike mathematic operations, the database engine eliminates **NULL** values from the calculations before performing the actual aggregations, at least at the most basic level. Let’s start with a simple **SELECT**statement:

|  |  |
| --- | --- |
|  | SELECT Name AS ProductName, Weight  FROM Production.Product  WHERE ProductID BETWEEN 715 AND 720; |

The statement returns the following results, which include two rows with a **Weight** value of **NULL**:

|  |  |
| --- | --- |
| **ProductName** | **Weight** |
| Long-Sleeve Logo Jersey, L | NULL |
| Long-Sleeve Logo Jersey, XL | NULL |
| HL Road Frame – Red, 62 | 2.30 |
| HL Road Frame – Red, 44 | 2.12 |
| HL Road Frame – Red, 48 | 2.16 |
| HL Road Frame – Red, 52 | 2.20 |

Suppose we now try to find the average weight for these rows:

|  |  |
| --- | --- |
|  | SELECT AVG(Weight)  FROM Production.Product  WHERE ProductID BETWEEN 715 AND 720; |

The database engine will perform the aggregation with no problem by first eliminating the **NULL** rows from the mix and then returning a result of **2.195000**. However, the results also include something else, a warning message about the **NULL** values:

|  |  |
| --- | --- |
|  | Warning: Null value is eliminated by an aggregate or other SET operation. |

We can verify that the **NULL** rows are being eliminated by instead using the **COUNT** aggregate function on the **Weight** column:

|  |  |
| --- | --- |
|  | SELECT COUNT(weight)  FROM Production.Product  WHERE ProductID BETWEEN 715 AND 720; |

This time the database engine returns a value of **4**, indicating that the two **NULL** rows have been eliminated. We will again receive the warning message.

All aggregate functions eliminate the **NULL** values prior to preforming the actual calculations except for the **COUNT(\*)** function:

|  |  |
| --- | --- |
|  | SELECT COUNT(\*)  FROM Production.Product  WHERE ProductID BETWEEN 715 AND 720; |

In this case, the database engine includes all rows, no matter where the **NULL** values lie, and will generate no warning messages.

The fact that SQL Server handles aggregations this way is not a bad thing. But it’s something you need to understand. If you’re not familiar with the data, your queries might be eliminating rows without understanding the full implications. You can consider using a function such as **ISNULL** to replace the **NULL**values, but you still risk skewing your results. As with many situations, handling **NULL** values can take a special touch. For example, detailed analytics will require, at the very least, knowledge of the number of **NULL** values in a particular data set.

As a side note, those who don’t appreciate having their queries return warning messages like those we’ve seen here, can use the **SET** **ANSI\_WARNINGS** statement to turn warnings off when executing an aggregation:

# Failure #11: Joining tables on nullable columns

Another area where we might get results other than what we expect is when we join data based on nullable columns. Because **NULL** value are considered non-values and because the database engine cannot match non-values, **NULL** values will show up in the join column only for outer joins.