Four Rules for NULLs

4 Simple Rules for Handling SQL NULLs

Introduction

"As we know,
There are known knowns.
There are things we know we know.
We also know
There are known unknowns.
That is to say
We know there are some things
We do not know."- Donald Rumsfeld

In a recent article by James Travis, the handling of NULLs in SQL Server was explored. I discovered some inaccuracies in the article; some caused by inaccuracies in SQL Server Books Online, others rooted in trying to apply other standard 3GL and 4GL programming language two-valued logic to the three-valued logic employed by ANSI SQL.

This article was written to explain ANSI SQL three-valued logic and correct the inaccuracies presented

What is NULL?

First, let's start by defining what a NULL is in SQL-speak. ANSI-SQL defines NULL as "a special value, or mark, that is used to indicate the absence of any data value." (ANSI-92) NULL in SQL is not a "data value", but rather an indicator that we have missing data. As an example, we'll look at the following table which contains last name, first name and middle name.

Last_Name	First_Name	Middle_Name
'Smith'	'Linda'	'Anita'
'Washington'	'Larry'	''
'Jones'	'John'	NULL

In this example, we know that Linda Smith has a middle name, and we know that it is 'Anita'. We have populated Larry Washington's Middle_Name column with a Zero-Length String ('') indicating that we know for a fact Larry has no middle name. These are the "known knowns" in Rumsfeld-speak.

In the case of John Jones, however, we have set the Middle_Name column to NULL. This indicates that we do not know whether John has a middle name or not. We may discover, in the future, that John has a middle name; likewise, we may discover he has no middle name. In either case, we will update his row accordingly. So, NULLs represent our "known unknowns"; the only thing we know for a fact is that we do *not* know whether John has a middle name or not.

Because of the inherent issues surrounding NULLs and data integrity, three-valued logic, scalar arithmetic and performance, DBA's and Database Designers tend to try to minimize the use of NULLABLE columns in their tables.

Rule #1: Use NULLs to indicate unknown/missing information only. Do not use NULLs in place of zeroes, zero-length strings or other "known" blank values. Update your NULLs with proper information as soon as possible.

Three-Valued Logic

Always in the Top 10 List of the most confusing aspects of ANSI SQL, for those coming from other 3GL and 4GL languages is three-valued logic. Anyone coming from a background in most other computer languages will instantly recognize the two-valued logic table:

Two-Valued Logic Table			
p	**q**	**p AND q**	**p OR q**
True	**True**	True	True
True	**False**	False	True
False	**True**	False	True
False	**False**	False	False

And of course the logic for NOT is fairly simple:

Two-Valued NOT Logic Table	
p	**NOT p**
True	False
False	True

So what is Three-Valued Logic? In SQL three-valued logic, we introduce the concept of Unknown into our logic, which is closely associated with NULL. Our SQL Logic Table looks like this:

Three-Valued Logic Table			
p	**q**	**p AND q**	**p OR q**
True	**True**	True	True
True	**False**	False	True
True	**Unknown**	Unknown	True
False	**True**	False	True
False	**False**	False	False
False	**Unknown**	False	Unknown
Unknown	**True**	Unknown	True
Unknown	**False**	False	Unknown
Unknown	**Unknown**	Unknown	Unknown

And in SQL logic, the NOT logic table looks like this:

Three-Valued NOT Logic Table

p	NOT p
True	False
False	True
Unknown	Unknown

So how do we arrive at Unknown in these logic tables? Simply put, we try to compare NULLs to data values, or other NULLs. Per the ANSI-92 SQL Standard, comparisons with NULL always result in Unknown.

Try the following SQL Query in Query Analyzer (requires the Northwind Sample Database):

```
SET ANSI_NULLS ON
SELECT * FROM [Northwind].[dbo].[Employees]
WHERE [Region] IS NULL
```

This query returns four employee records. Now try this query:

```
SET ANSI_NULLS ON
SELECT * FROM [Northwind].[dbo].[Employees]
WHERE [Region] = NULL
```

This query returns no records. The reason for this is that '[Region] = NULL' evaluates to Unknown every time. Since it never evaluates to True for any rows, it returns no records.

NOTE: It has been widely misstated (even in SQL Server Books Online) that "the result of a comparison is false if one of the operands is NULL." In fact, the ANSI-92 SQL Standard states that if one of the operands is NULL, the result of the comparison is "UNKNOWN". (ANSI-92 SQL Standard) SQL Books Online has recently been updated to reflect the accurate ANSI Standard concerning NULL comparisons (See the update here: Microsoft ANSI_NULLS).

NULL Versus NULL

NULLs are not equal to other NULLs. In his article, Understanding the Difference Between "IS NULL" and "=NULL", James Travis states that a comparison to a variable initialized to NULL returns False. And it would appear so by his example:

```
DECLARE @val CHAR(4)
SET @val = NULL
SET ANSI_NULLS ON
If @val = NULL
     PRINT 'TRUE'
ELSE
     PRINT 'FALSE'
```

Mr. Travis' example prints 'FALSE', which indicates that the result of the comparison '@val = NULL' is False. This is not the case, however. The result of '@val = NULL' is Unknown. The IF statement will print 'TRUE' if, and only if, the result of the comparison is True; otherwise it falls through to the ELSE statement and prints 'FALSE'. For three-valued logic, we need three comparisons to figure out what the actual result of the '@val = NULL' comparison is:

```
SET ANSI_NULLS ON
DECLARE @val CHAR(4)
SET @val = NULL
SET ANSI_NULLS ON
If @val = NULL
```

```
     PRINT 'TRUE'
ELSE IF NOT(@val = NULL)
     PRINT 'FALSE'
ELSE
     PRINT 'UNKNOWN'
```

Now we see that the result of the comparison is neither True nor False. It is the third value in our three-valued logic, Unknown. Try this statement to demonstrate that NULL is not equal to other NULLs:

```
SET ANSI_NULLS ON
IF NULL = NULL
     PRINT 'TRUE'
ELSE IF NOT (NULL = NULL)
     PRINT 'FALSE'
ELSE
     PRINT 'UNKNOWN'
```

This statement prints 'UNKNOWN' because 'NULL = NULL' evaluates to Unknown. Since 'NULL = NULL' evaluates to Unknown, if we look back at the NOT Logic table for three-valued logic, we see that 'NOT(Unknown)' also evaluates to Unknown. This means that 'NOT(NULL = NULL)' is Unknown as well. So what do we expect when we run this statement?

```
SET ANSI_NULLS ON
IF NULL <> NULL
     PRINT 'TRUE'
ELSE IF NOT (NULL <> NULL)
     PRINT 'FALSE'
ELSE
     PRINT 'UNKNOWN'
```

If you expect to see ‘TRUE’, guess again! ‘FALSE’? Not a chance. Remember what we said earlier – any comparisons with NULL evaluate to Unknown (neither True nor False). Since the comparisons in the IF statements evaluate to Unknown, ‘UNKNOWN’ is printed in this instance.

The same is true for any comparisons with NULL under the ANSI Standard; whether you are comparing NULL to a CHAR, INT or any other value, variable or table column.

Rule #2: In ANSI SQL, NULL is not equal to anything, even other NULLs! Comparisons with NULL always result in UNKNOWN.

NULL Variables

Variables that have been declared, but not yet initialized via SET statement default to NULL. They are treated exactly like any other NULL values. We can modify the previous queries to see this for ourselves:

```
SET ANSI_NULLS ON
DECLARE @MyRegion NVARCHAR(15)
SELECT * FROM [Northwind].[dbo].[Employees]
WHERE [Region] = @MyRegion
```

This returns no rows, just like in the previous query with '=NULL' in the WHERE clause. This has exactly the same effect as the following:

```
SET ANSI_NULLS ON
DECLARE @MyRegion NVARCHAR(15)
SET @MyRegion = NULL
```

```
SELECT * FROM [Northwind].[dbo].[Employees]
WHERE [Region] = @MyRegion
```

NOTE: You cannot put a variable in place of NULL in the IS NULL clause. The following query will *not* work:

```
SET ANSI_NULLS ON
DECLARE @MyRegion NVARCHAR(15)
SELECT * FROM [Northwind].[dbo].[Employees]
WHERE [Region] IS @MyRegion
```

ANSI Versus Microsoft

As we described above, the ANSI SQL Standard describes the result of any comparison with NULL as Unknown. So NULL = NULL, NULL <> 10 and NULL < 'Hello' are all Unknown.

Presumably to make the transition smoother for 3GL and 4GL programmers from other languages moving over to SQL, Microsoft implemented the SET ANSI_NULLS OFF option, which allows you to perform = and <> comparisons with NULL values. To demonstrate this in action, we can perform the following query:

```
SET ANSI_NULLS OFF
SELECT * FROM [Northwind].[dbo].[Employees]
WHERE [Region] = NULL
```

This allows you to perform basic equality comparisons to NULL; however, SQL written using this option may not be portable to other platforms, and might confuse others who use ANSI Standard NULL-handling. Also, if you try to perform this query without turning ANSI_NULLS OFF first, you will get no rows returned. The safe bet is to use SET ANSI_NULLS ON.

Rule #3: Use SET ANSI_NULLS ON, and always use ANSI Standard SQL Syntax for NULLs. Straying from the standard can cause problems including portability issues, incompatibility with existing code and databases and returning incorrect results.

NULL Math

One final point concerns math and string operations with NULLs. If you perform scalar math operations and string concatenation functions with NULL, the result is always NULL. For instance, this query returns NULL:

```
SELECT 1 + NULL
```

A numeric value plus, minus, divided by, or multiplied by NULL always equals NULL. A string concatenation with NULL equals NULL as well:

```
SELECT 'Joe' + NULL
```

So what if you want to treat NULL as a zero in a math operation? Use SQL's COALESCE() function:

```
SELECT 1 + COALESCE(NULL, 0)
```

The COALESCE() function returns the first non-NULL value in its list of values. So in the statement above, "COALESCE(NULL, 0)" returns 0. The SELECT statement above returns 1. Similarly you can use COALESCE() in string concatenations:

```
SELECT 'Joe' + COALESCE(NULL, '')
```

The COALESCE() function in this case returns a Zero-Length String, which is appended to 'Joe'. The end result is that 'Joe' is returned instead of NULL. SQL Server provides an additional function called ISNULL() which performs similarly to COALESCE(). Use the COALESCE() function (or alternatively use CASE), as it is ANSI standard syntax.

Rule #4: The ANSI Standard COALESCE() and CASE syntaxes are preferred over ISNULL() or other proprietary syntax.

Conclusions

SQL NULLs and three-valued logic can be confusing for a new SQL Database Designer; particularly if you are coming from a background in another 3GL or 4GL language that uses two-valued logic (i.e., C++, VB, etc.) Learning when and how to use NULLs in your databases, how SQL handles NULLs, and the results of comparisons and math/string operations with NULLs in SQL is very important in order to obtain consistent and optimized results.

For further information, you can visit:

- Microsoft SQL Server MSDN Website
- American National Standards Institute