NULL versus NULL

The following article was taken from: http://www.sqlservercentral.com/articles/Advanced+Querying/2829/. Because some of the code listings are not fully visible, they are reproduced below.

Listing #5

```
CREATE TABLE #test (val INT CONSTRAINT unq_val UNIQUE);

INSERT INTO #test (val) VALUES (NULL);
INSERT INTO #test (val) VALUES (NULL);

(1 row(s) affected)
Msg 2627, Level 14, State 1, Line 4
Violation of UNIQUE KEY constraint 'unq_val'. Cannot insert duplicate key in
object 'dbo.#test'.
The statement has been terminated.
```

Listing #7

```
CREATE TABLE #test (val INT CONSTRAINT ck_val CHECK(val < 0 AND val = 0 AND val > 0));

INSERT INTO #test (val) VALUES (NULL);
INSERT INTO #test (val) VALUES (NULL);

SELECT val
FROM #test
ORDER BY val;

DROP TABLE #test;
```

NULL Versus NULL?

By Michael Coles, 2007/02/26

NULL Versus NULL

In one of the first articles I wrote for SQL Server Central, I talked about SQL NULLs and three-valued logic (Four Rules For NULL). In this article I take it all back...

No, not really, but stay tuned as we talk about the darker side of ANSI NULLs.

The Original Four Rules

The original four rules I proposed for NULL-handling are all reproduced here in Figure 1.

Figure 1. The original "Four Rules"

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.
Rule #2	In ANSI SQL, NULL is not equal to anything, even other NULLs. Comparisons with NULL always result in UNKNOWN.
Rule #3	Use SET ANSI_NULLS ON, and always use ANSI Standard SQ Syntax for NULLs. Straying from the standard can cause prob including portability issues, incompatibility with existing code databases and returning incorrect results.
Rule #4	The ANSI Standard COALESCE() and CASE syntaxes are preferred over ISNULL() or other proprietary syntax.

Zoom in | Open in new window

The rules are handy guides for handling NULLs in T-SQL, but NULL-handling isn't always so cut-and-dried. In this article we'll take a look at where Rule #2 - the basis of the ANSI SQL three-valued logic we discussed in the first article - breaks down.

No Two NULLs Are Created Equal...

If you recall from the original Four Rules article, the basis of ANSI SQL three-valued logic (3VL) is that NULL is not equal to anything else. It is not less than, greater than, or even unequal to anything else either. Because NULL is not an actual value, but rather a placeholder for an unknown value, all comparisons with NULL result in UNKNOWN. Even comparing a NULL to another NULL is just comparing two placeholders for unknown values, so the result again is UNKNOWN.

> **Tip:** In reference to NULL comparisons, be sure to keep Rule #3 in mind. Microsoft has deprecated SET ANSI_NULLS, and according to Books Online it will be removed in a future version of SQL Server. If you currently have code that relies on SET ANSI_NULLS OFF, it might be a good time to start considering what it will take to make that code ANSI SQL-92 NULL-compliant.

We even generated some samples to demonstrate this. One of these samples is reproduced here in Listing 1.

Listing 1. Demonstrating that NULL is not equal to NULL

```
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'
```

NULL: Confusing the Smartest People in the World Since (at least) 1986

If all this doesn't hit home immediately, don't take it too hard. Even Microsoft seems to have difficulty sorting through it. Point in fact: SQL Server 2005 Books Online (BOL) still has bad information concerning `NULL` comparisons. In fact, as of the time of this writing I counted no less than ten pages in BOL that stated the result of a comparison with `NULL` is either FALSE or `NULL`. Only two pages that I found (the pages describing "`IS [NOT] NULL`" and "`SET ANSI_NULLS`") actually got it right. Fortunately we know better: the result of comparing `NULL` with anything is UNKNOWN.

So why all the confusion? Most likely it's because in queries only rows for which the `WHERE` clause condition evaluates to TRUE are returned. Rows that evaluate to FALSE or UNKNOWN are not returned. For some folks this might seem to indicate FALSE and UNKNOWN are equivalent. They're not, as we'll see in Listings 2 and 3.

Listing 2. Sample SELECT with NULL comparison in the WHERE clause

```
SELECT TOP 100 *
FROM sys.syscomments
WHERE id = NULL
```

The result above of course returns no rows. According to Books Online this is because "id = NULL" evaluates to FALSE. If this is true, however, Listing 3 below should return **all** rows.

Listing 3. The "opposite" of Listing 2

```
SELECT TOP 100 *
FROM sys.syscomments
WHERE NOT(id = NULL)
```

If "`id = NULL`" really evaluates to FALSE for every row, then "`NOT (id = NULL)`" should evaluate to TRUE for every row. Of course it doesn't, and again no rows are returned. And we already know the reason: it's because "`id = NULL`" evaluates to UNKNOWN, and "`NOT(id = NULL)`" also evaluates to UNKNOWN.

Microsoft has already been notified of this problem in BOL and hopefully it will be fixed soon.

...All NULLs Are Created *Not Distinct*

So now we've firmly established that comparisons with `NULL` never evaluate to TRUE or FALSE, that `NULL` is never equal to `NULL`, and that `NULL` comparisons always result in UNKNOWN... Now it's time to list the *exceptions*. (You didn't think it would be that simple did you?)

I really wanted to call this section *All NULLs Are Created Equal*, but that just happens to be wrong. In order to *simulate* `NULL` equality, and to keep from contradicting themselves in the process, the ANSI SQL-92 standard decreed that two `NULL` values should be considered "*not distinct*". The definition of *not distinct* in the ANSI standard includes any two values that return TRUE for an equality test (e.g., 3 = 3, 4 = 4, etc.), or any two `NULL`s.

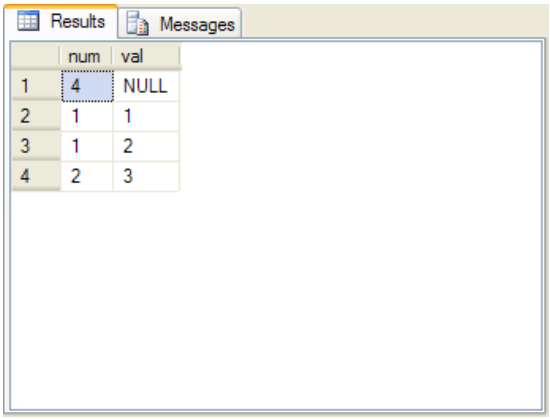
This simulated `NULL` equality is probably most used in the `GROUP BY` clause, which groups all `NULL` values into a single *partition*. SQL-92 defines a *partition* as a grouping of *not distinct values*. Listing 4 below shows `GROUP BY` handling of `NULL`.

Listing 4. GROUP BY and NULL

```
CREATE TABLE #test (val INT);

INSERT INTO #test (val) VALUES (NULL);
INSERT INTO #test (val) VALUES (NULL);
INSERT INTO #test (val) VALUES (NULL);
INSERT INTO #test (val) VALUES (NULL);
INSERT INTO #test (val) VALUES (1);
INSERT INTO #test (val) VALUES (2);
INSERT INTO #test (val) VALUES (3);
INSERT INTO #test (val) VALUES (3);

SELECT COUNT(*) AS num, val
FROM #test
GROUP BY val;

DROP TABLE #test;
```

Figure 2 shows the result.

Figure 2. Result of GROUP BY with NULLs



Notice the `NULL` values are all treated as *not distinct* by `GROUP BY`, and are all grouped together. Unique constraints also use the ANSI definition of *not distinct* as opposed to *equal* since you can only insert one `NULL` in a column with a unique constraint. Consider Listing 5 which shows this.

Listing 5. Unique Constraint and NULL

```
CREATE TABLE #test (val INT CONSTRAINT unq_val

INSERT INTO #test (val) VALUES (NULL);
INSERT INTO #test (val) VALUES (NULL);
```

This example throws an exception when it tries to insert the second `NULL` in the `val` column:

```
(1 row(s) affected)
Msg 2627, Level 14, State 1, Line 4
Violation of UNIQUE KEY constraint 'unq_val'.
object 'dbo.#test'.
```

Other statements and operators that use the concept of *not distinct* to simulate `NULL` equality include:

- `PARTITION BY` clause of `OVER()`
- `UNION` operator
- `DISTINCT` keyword
- `INTERSECT` operator
- `EXCEPT` operator

NULLs Flock Together

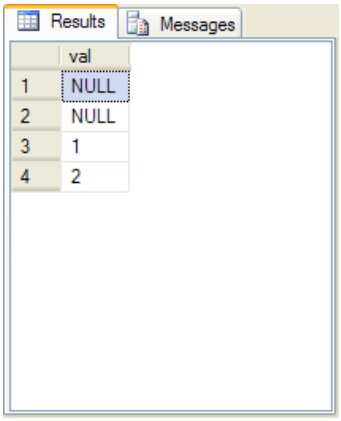
The `ORDER BY` clause in `SELECT` queries places all `NULL` values together when it orders your results. SQL Server treats `NULL`s as the "lowest possible values" in your results. What this means is `NULL` will always come before your non-`NULL` results when you sort in ascending order, and after your non-`NULL` results when you sort in descending order. Listing 6 shows `ORDER BY` and `NULL` in action.

Listing 6. ORDER BY and NULL

```
CREATE TABLE #test (val INT);

INSERT INTO #test (val) VALUES (NULL);
INSERT INTO #test (val) VALUES (NULL);
INSERT INTO #test (val) VALUES (1);
INSERT INTO #test (val) VALUES (2);

SELECT val
FROM #test
ORDER BY val;

DROP TABLE #test;
```

The results are shown in Figure 3.

Figure 3. Result of ORDER BY with NULL



The same holds true for the `ORDER BY` clause of `OVER`, which is used to order your results when used with ranking functions like `ROW_NUMBER` and aggregate functions like `SUM`.

And Now For Something Entirely Different

Now that we've established the "exceptions" for `NULL` comparisons, let's look at something entirely different. When a `NULL` value is inserted into a nullable column with a check constraint that doesn't check for `IS NOT NULL`, something strange seems to happen. Consider Listing 7.

Listing 7. Check constraints and NULL

```
CREATE TABLE #test (val INT CONSTRAINT ck_val

INSERT INTO #test (val) VALUES (NULL);
INSERT INTO #test (val) VALUES (NULL);

SELECT val
FROM #test
ORDER BY val;
```

In this example we've added a check constraint to the sample table that enforces the following rule:

- The value inserted must be less than zero
- *and* the value inserted must be equal to zero
- *and* the value inserted must be greater than zero

You and I know from 4th grade math (remember number lines?) that there is no value that can ever fulfill these requirements. No value can be less than zero, equal to zero, and greater than zero all at the same time. Also based on what we've already talked about, any comparisons with `NULL` result in UNKNOWN. You might expect an attempt to insert any value into the table would fail.

However, check constraints operate under a different set of rules from the `SELECT`, `INSERT`, `UPDATE`, and `DELETE` DML statements. The DML statements, when combined with a `WHERE` clause, perform their action only on rows for which the `WHERE` clause condition evaluates to TRUE. The DML statements will exclude rows that evaluate to FALSE or UNKNOWN.

Check constraints, on the other hand, cause `INSERT` and `UPDATE` statements to fail only if the check constraint condition evaluates to FALSE. This means that the checks will succeed if the condition evaluates to either UNKNOWN or TRUE.

Of course you'd probably never create a check constraint as restrictive as the one in the example, and if you want to prevent `NULL`s from being inserted into a column, either declare the column `NOT NULL` or add "`val IS NOT NULL`" as a check constraint condition. Don't expect a check constraint that evaluates to UNKNOWN to cause an `INSERT` or `UPDATE` to fail.

Conclusion

`NULL` handling hasn't gotten any easier since the Four Rules article, but it helps to know the exceptions as well as the rules. This article was written to demonstrate those common exceptions.

Michael Coles is a regular contributor to SQL Server Central, and author of the upcoming book *Pro T-SQL 2005 Programmer's Guide* from Apress (in bookstores everywhere April 2007).

By Michael Coles, 2007/02/26