Behavior of Nulls

Scott Peters

www.advancedsqlpuzzles.com

Last updated 05/01/2021

Behavior of Nulls

NULL markers are a source of much confusion and trouble in SQL and they should be understood well to avoid any errors. To best understand NULL markers, one must recognize how they are treated in the different elements of the SQL language. The JOIN syntax will treat NULL markers differently then SET OPERATORS, and a UNIQUE constraint will treat a NULL marker differently than a PRIMARY KEY constraint. Here in the following pages we will look at various SQL constructs and determine how the NULL marker is treated.

Notice I write NULL marker, and not NULL value. NULL is not a data value, but a marker representing the absence of a value. Simply put, NULL represents an unknown value. It is worth noting that in relational algebra, NULL markers are categorized into two distinct groups, NULL markers that have a missing value but are applicable (A-Mark), and NULL markers whose values are both missing and inapplicable (I-Mark). An example of an A-Mark could be someone's birthday, or their blood type. Everyone has a birthdate and a blood type, but these values maybe unknown to some people. An example of an I-Mark would be an ordering system where dummy orders are placed to reconcile actual inventory to inventory sold. The Customer ID of these orders would be NULL, as no actual Customer ID exist for these transactions. This is one of the ways relational algebra differs from SQL.

The following document is based on Microsoft SQL Server 2014. I would be happy to receive corrections, additions, new tricks and techniques, and other suggestions at scottpeters1188@outlook.com

The latest version of this document can be found at www.advancedsqlpuzzles.com

Sample Data

First, let's create the following tables in our quest to understand the behavior of NULLS. We will work with this data through out these exercises.

```
IF OBJECT_ID('tempdb.dbo.##Nulls1','U') IS NOT NULL
  DROP TABLE #Nulls1;
GO

IF OBJECT_ID('tempdb.dbo.##Nulls2','U') IS NOT NULL
  DROP TABLE #Nulls2;
GO

CREATE TABLE #Nulls1
(
NullField INTEGER,
TextField VARCHAR(10)
);
GO

CREATE TABLE #Nulls2
(
NullField INTEGER,
TextField VARCHAR(10)
);
GO

INSERT INTO #Nulls1 VALUES
(NULL,'Apples'),(NULL,'Oranges'),(1,'Bananas');

INSERT INTO #Nulls2 VALUES
(NULL,'Apples'),(NULL,'Oranges'),(1,'Bananas'),(2,'Grapes');
```

```
IF OBJECT_ID('tempdb.dbo.##NullConstraints','U') IS NOT NULL
  DROP TABLE ##NullConstraints;
GO

CREATE TABLE ##NullConstraints
(
MyField1 INTEGER,
MyField2 INTEGER,
MyField3 INTEGER,
MyField4 INTEGER
);
GO
```

PREDICATE LOGIC

To best understand NULL markers in SQL, you need to understand the three-valued logic outcomes of TRUE, FALSE, and UNKNOWN. While NOT TRUE is FALSE, and NOT FALSE is TRUE, the opposite of UNKNOWN will always be UNKNOWN. Unique to SQL, the logic result will always be UNKNOWN when comparing a NULL marker to any other value. SQL's use of the three-valued logic system presents a surprising amount of complexity into a seemingly straightforward query.

The following truth tables display how the three-valued logic is applied.

AND	**TRUE**	**UNKNOWN**	**FALSE**
TRUE	TRUE	UNKNOWN	FALSE
UNKNOWN	UNKNOWN	UNKNOWN	FALSE
FALSE	FALSE	FALSE	FALSE

OR	**TRUE**	**UNKNOWN**	**FALSE**
TRUE	TRUE	TRUE	TRUE
UNKNOWN	TRUE	UNKNOWN	UNKNOWN
FALSE	TRUE	UNKNOWN	FALSE

NOT	
TRUE	FALSE
UNKNOWN	UNKNOWN
FALSE	TRUE

A good example of the complexity is shown below in the following example.

TRUE OR UNKNOWN = TRUE

```
SELECT 1 WHERE ((1=1) OR (NULL=1));
```

FALSE OR UNKNOWN = UNKNOWN

```
SELECT 1 WHERE NOT((1=2) OR (NULL=1));
```

Because TRUE OR UNKNOWN equates to TRUE, one may expect NOT(FALSE OR UNKNOWN) to equate to TRUE, but this is not the case. UNKNOWN could potentially have the value of TRUE or FALSE, leading to the second statement to either be TRUE or FALSE if the value of UNKNOWN becomes available.

The statement TRUE OR UNKNOWN will always resolve to TRUE if the value of UNKNOWN is either TRUE or FALSE, as TRUE OR FALSE is always TRUE.

ANSI_NULLS

In SQL Server, the SET ANSI_NULLS setting specifies the ISO compliant behavior of the Equals (=) and Not Equal To (<>) comparison operators. The following table shows how the setting of ANSI_NULLS affects the results of Boolean expressions using NULL markers. Note the standard setting for ANSI_NULLS is ON.

Boolean Expression	SET ANSI_NULLS ON	SET ANSI_NULLS OFF
NULL = NULL	UNKNOWN	TRUE
1 = NULL	UNKNOWN	FALSE
NULL <> NULL	UNKNOWN	FALSE
1 <> NULL	UNKNOWN	TRUE
NULL > NULL	UNKNOWN	UNKNOWN
1 > NULL	UNKNOWN	UNKNOWN
NULL IS NULL	TRUE	TRUE
1 IS NULL	FALSE	FALSE
NULL IS NOT NULL	FALSE	FALSE
1 IS NOT NULL	TRUE	TRUE

Here we can experiment with setting the ANSI_NULLS setting. Feel free to set the ANSI_NULLS to ON and OFF to see how the behavior changes.

SQL provides the IS NULL and IS NOT NULL operators to determine if a specified expression is NULL.

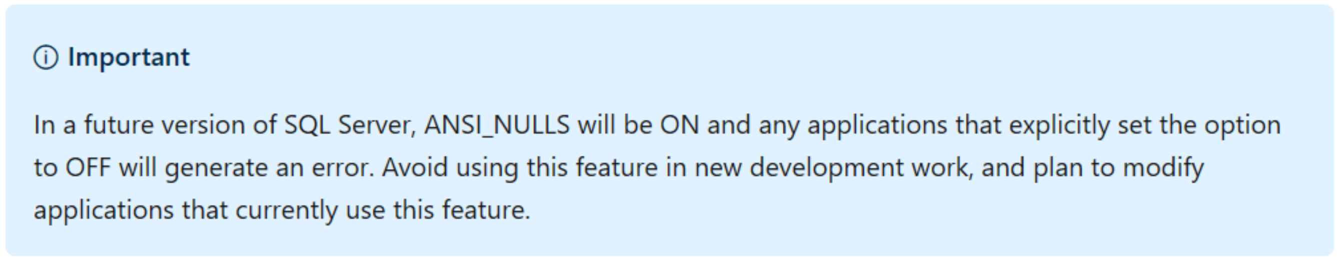
```
SET ANSI_NULLS ON;

--UNKNOWN
SELECT 1 WHERE NULL = NULL;
SELECT 1 WHERE 1 = NULL;
SELECT 1 WHERE NULL <> NULL;
SELECT 1 WHERE NULL = NULL;
SELECT 1 WHERE 1 = NULL;
SELECT 1 WHERE NULL <> NULL;
SELECT 1 WHERE 1 <> NULL;
SELECT 1 WHERE NULL > NULL;
SELECT 1 WHERE 1 > NULL;

--TRUE
SELECT 1 WHERE NULL IS NULL;
SELECT 1 WHERE 1 IS NOT NULL;

--FALSE
SELECT 1 WHERE 1 IS NULL;
SELECT 1 WHERE NULL IS NOT NULL;
```

In the following examples we will set the default ANSI_NULLS setting to ON as the ANSI NULLS will be removed in future versions of SQL server.



ⓘ **Important**

In a future version of SQL Server, ANSI_NULLS will be ON and any applications that explicitly set the option to OFF will generate an error. Avoid using this feature in new development work, and plan to modify applications that currently use this feature.

https://docs.microsoft.com/en-us/sql/t-sql/statements/set-ansi-nulls-transact-sql?view=sql-server-ver15

JOIN SYNTAX

Let's look at how the JOIN syntax interprets NULLS.

If you haven't done so, create the sample data provided here that creates the below two tables.

#Nulls1

NullField	TextField
<NULL>	Apples
<NULL>	Oranges
1	Bananas

#Nulls2

NullField	TextField
<NULL>	Apples
<NULL>	Oranges
1	Bananas
2	Grapes

From the following two SQL statements we can see that the JOIN syntax does not treat the NULL markers as equalities or inequalities, but as UNKNOWNS.

The first query uses an equi join on the NullField column. The NULL markers are eradicated as they are treated as UNKNOWNS.

```
SELECT *
FROM   #Nulls1 a INNER JOIN
       #Nulls2 b ON a.NullField = b.NullField;
```

NullField	TextField	NullField	TextField
1	Bananas	1	Bananas

The second query here uses a theta join on the NullField. Like the equi join above, the NULLS are eradicated as they are treated as UNKNOWNS.

```
SELECT *
FROM   #Nulls1 a INNER JOIN
       #Nulls2 b ON a.NullField <> b.NullField and a.TextField = b.TextField;
```

NullField	TextField	NullField	TextField

The result set returns 0 rows.

SEMI AND ANTI JOINS

Simply put, semi-joins use the IN or EXISTS operator. Anti-joins use the NOT IN or NOT EXISTS operator. Please refer to your favorite relational algebra textbook for a more formal definition!

Here we will go one by one and demonstrate their behavior.

IN

The IN operator will return a result set.

```
SELECT 'Hello World'
WHERE 2 IN (NULL,2);
```

NOT IN

The NOT IN will return a NULL result set. NULL could potentially resolve to any number, rendering this statement UNKNOWN.

```
SELECT 'Hello World'
WHERE 1 NOT IN (NULL,2);
```

EXISTS

The following example returns a result set with NULL specified in the subquery.

```
SELECT *
FROM   #Nulls1 a
WHERE  EXISTS(SELECT NULL);
```

This statement will ignore the NULL markers and return the value Bananas.

```
SELECT *
FROM   #Nulls1 a
WHERE  EXISTS(SELECT NullField FROM #Nulls2 b WHERE a.NullField = b.NullField);
```

DOES NOT EXIST

This statement will return Apples and Oranges in the result set, as it treats the NULL markers as inequalities.

```
SELECT *
FROM   #Nulls1 a
WHERE  NOT EXISTS(SELECT NullField FROM #Nulls2 b
                    WHERE a.NullField = b.NullField);
```

ANY / ALL OPERATORS

Two often overlooked operators are the ANY and ALL operators.

ANY

This correlated subquery can be rewritten with the ANY operator and produce the same results.

```
SELECT *
FROM   #Nulls1 a
WHERE  EXISTS(SELECT NullField FROM #Nulls2 b WHERE a.NullField = b.NullField);

SELECT *
FROM   #Nulls1 a
WHERE  a.NullField = ANY(SELECT b.NullField FROM #Nulls2 b);
```

ALL

The below statement will evaluate to NULL, as we are unsure if the integer 5 is equal to or not equal to a relation that contains a NULL marker.

```
SELECT 'Hello World'
WHERE  5 <> ALL(SELECT 1 UNION SELECT 2 UNION SELECT NULL);
```

SET OPERATORS

The SQL standard for SET OPERATORS does not use the term *EQUAL TO* and *NOT EQUAL TO* when describing their behavior. Instead it uses the terminology of *IS [NOT] DISTINCT FROM* and the following expressions are TRUE when using SET OPERATORS. NULL is not distinct from NULL, NULL is distinct from Apples.

In the following example, we see the SET OPERATORS treat the NULL markers differently than the JOIN syntax.

```
SELECT NullField FROM #Nulls1
UNION ALL
SELECT NullField FROM #Nulls2;
```

NullField
NULL
NULL
1
NULL
NULL
1
2

```
SELECT NullField FROM #Nulls2
EXCEPT
SELECT NullField FROM #Nulls1;
```

NullField
2

```
SELECT * FROM ##Nulls1
INTERSECT
```

```
SELECT * FROM ##Nulls2;
```

NullField
NULL
1

GROUP BY

Here the GROUP BY syntax sees the NULL markers as equals and aggregates them together.

```
SELECT NullField, COUNT(*) AS COUNT
FROM   ##Nulls1
GROUP BY NullField;
```

NullField	COUNT
NULL	2
1	1

COUNT FUNCTION

Here the COUNT function removes the NULL markers when performed on a specific field, but counts the NULL markers when using the *.

```
SELECT COUNT(NullField) AS CountNullField,
       COUNT(*) AS CountStar,
       IIF(COUNT(*) <> COUNT(NullField),'Not Equals','Equals')  AS CountComparison
FROM ##Nulls1;
```

CountNullField	CountStar	CountComparison
1	3	Not Equals

CONSTRAINTS

Now let's look at how constraints treat the NULL marker.

If you haven't done so, create the sample data provided here.

PRIMARY KEYS

First, note that a PRIMARY KEY will create a CLUSTERED INDEX unless specified otherwise, and a UNIQUE constraint will create a NONCLUSTERED INDEX unless specified otherwise.

The following statements will error as a PRIMARY KEY does not allow for NULL markers.

```
ALTER TABLE ##NullConstraints
ADD CONSTRAINT PK_NullConstraints PRIMARY KEY NONCLUSTERED (MyField1);

ALTER TABLE ##NullConstraints
ADD CONSTRAINT PK_NullConstraints PRIMARY KEY CLUSTERED (MyField1);
```

The error statement produced will be.

```
“Cannot define PRIMARY KEY constraint on nullable column in table
'##NullConstraints'.”
```

UNIQUE Constraints

Let’s move on and create these constraints.

```
ALTER TABLE ##NullConstraints
ADD CONSTRAINT UNIQUE_NullConstraints UNIQUE (MyField2);
GO

INSERT INTO ##NullConstraints(MyField2) VALUES (NULL);
GO
INSERT INTO ##NullConstraints(MyField2) VALUES (NULL);
GO
```

Here we notice that only one NULL marker is allowed in a UNIQUE constraint. The second statement produces the following error.

```
“Violation of UNIQUE KEY constraint 'UNIQUE_NullConstraints'. Cannot insert
duplicate key in object 'dbo.##NullConstraints'. The duplicate key value is
(<NULL>).”
```

CHECK Constraints

First run a truncate table so the above inserted NULL marker will not cause a problem. Then create the following two CHECK constraints.

```
TRUNCATE TABLE ##NullConstraints;
GO
ALTER TABLE ##NullConstraints
ADD CONSTRAINT CHECK_NullConstraints1 CHECK (MyField3 > 0);
GO
ALTER TABLE ##NullConstraints
ADD CONSTRAINT CHECK_NullConstraints2 CHECK (MyField4 < 0);
GO

INSERT INTO ##NullConstraints (MyField3,MyField4) VALUES (NULL,NULL);
GO
```

The above insert does not error and allows for the insert to take place.

Referential Integrity

In SQL Server, a foreign key constraint must be linked to a column with either a PRIMARY KEY constraint or a UNIQUE constraint defined on the column. Remember that a PRIMARY KEY constraint does not allow NULL markers, but a UNIQUE constraint allows one NULL marker.

Here in the below example we can demonstrate that multiple NULL markers can be inserted into a child column that references another column only if the reference column has a unique constraint on the table. Multiple NULL markers can be inserted into the child column.

```
IF OBJECT_ID('dflt.ParentChild') IS NOT NULL
       DROP TABLE dflt.ParentChild

CREATE TABLE dflt.ParentChild
(
 RowID_PrimaryKey INTEGER PRIMARY KEY
,RowID_Unique INTEGER UNIQUE
,RowID_NoConstraint INTEGER
,ChildID_1 INTEGER FOREIGN KEY REFERENCES dflt.ParentChild (RowID_PrimaryKey)
,ChildID_2 INTEGER FOREIGN KEY REFERENCES dflt.ParentChild (RowID_Unique)
/*,ChildID_3 INTEGER FOREIGN KEY REFERENCES dflt.ParentChild (RowID_NoConstraint)*/
);

INSERT INTO dflt.ParentChild
(RowID_PrimaryKey, RowID_Unique, RowID_NoConstraint, ChildID_1, ChildID_2)
VALUES (1,2,3,1,2);

INSERT INTO dflt.ParentChild
(RowID_PrimaryKey, RowID_Unique, RowID_NoConstraint, ChildID_1, ChildID_2)
VALUES (2,NULL,3,1,NULL);

INSERT INTO dflt.ParentChild
(RowID_PrimaryKey, RowID_Unique, RowID_NoConstraint, ChildID_1, ChildID_2)
VALUES (3,4,5,1,NULL);
```

Computed Columns

A computed column is a virtual column that is not physically stored in a table. A computed column expression can use data from other columns to calculate a value. When an expression is applied to a column with a NULL marker, a NULL marker will be the return value.

Here we demonstrate that behavior.

```
WITH cte_ColumnExpression AS
(SELECT NULL AS MyInteger
UNION
SELECT 100)
SELECT MyInteger * 2
FROM   cte_ColumnExpression;
```

MyExpression
NULL
200

SQL Functions

SQL provides three functions to help evaluate NULL markers; ISNULL, NULLIF, and COALESCE.

ISNULL

The ISNULL function replaces NULL with the specified replacement value.

```
ISNULL ( check_expression , replacement_value)
```

NULLIF

The NULLIF returns a NULL marker if the two specified expressions are equal.

```
NULLIF ( expression , expression)
```

COALESCE

The COALESCE function evaluates the arguments in order and returns the current value of the first expression that does not evaluate to NULL.

```
COALESCE ( expression [ ,…n ] )
```

The Empty String

A useful feature to combat NULL markers in character fields is by using the empty string, which is simply the character ''.

Note that this character is not an ASCII value and the following function returns NULL. `SELECT ASCII('');`

Any queries that join on a field with an empty string will equate to true. Commonly the empty string is used with the ISNULL function, such as the following query.

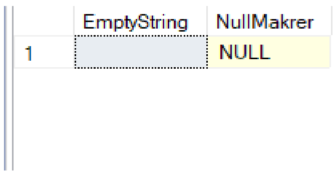
```
SELECT *
FROM   #Nulls1 a INNER JOIN
       #Nulls2 b ON ISNULL(a.NullField,'') = ISNULL(b.NullField,'');
```

If you create a table with the empty string, the output can deceivingly look like a NULL marker. Here we will view the output of a NULL marker and the empty string in SSMS.

```
CREATE TABLE #EmptyString
(
EmptyString VARCHAR(100),
NullMakrer    VARCHAR(100)
);
GO

INSERT INTO #EmptyString VALUES ('',NULL);
GO

SELECT * FROM #EmptyString;
```



```
   EmptyString  NullMakrer
1               NULL
```

Note that the NULL marker is stated as “NULL” and the field is highlighted in yellow. The Empty String is simply blank.

Conclusion

I hope you learned a few new things about NULL markers and can use this document as a guide when addressing the behavior of NULLs. I often find myself forgetting exactly how NULLs are treated within the database, which led me to create this document in the first place.

If you find any errors, think I am omitting anything, or just want to say hello, please email me at scottpeters1188@outlook.com.