BEHAVIOR OF NULLS

Scott Peters

https://advancedsqlpuzzles.com

Last updated: 04/08/2022

To record missing or unknown values, users of relational databases can assign NULL markers to columns. NULL is not a data value, but a marker representing the absence of a value.

NULL markers can mean one of two things:

- the column does not apply to the other columns in the record
- the column applies, but the information is unknown.

Because NULL markers represent the absence of a value, NULL markers can be a source of much confusion and trouble to developers. To best understand NULL markers, one must understand the three-valued logic of true, false, or unknown, and recognize how the NULL markers are treated within the different constructs of the SQL language. The join syntax will treat NULL markers differently than set operators, and a unique constraint will treat a NULL marker differently than a primary key constraint.

Because NUL markers do not represent a value, SQL has two conditions specific to the SQL language:

- IS NULL
- IS NOT NULL

SQL also provides three functions to evaluate NULL markers:

- NULLIF
- ISNULL
- COALESCE.

We will cover these aspects and many more in the following document.

The SQL statements and diagrams from this document can be found in following Git repository.
https://github.com/smpetersgithub/AdvancedSQLPuzzles/tree/main/Database%20Writings

The examples are based on Microsoft's SQL Server. The provided SQL statements can be modified easily to fit any dialect of SQL.

I welcome any corrections, new tricks, new techniques, dead links, misspellings, bugs, and especially any new puzzles that would be a great fit for this document.

Please contact me through the contact page on my website.

PREDICATE LOGIC

To best understand NULL markers in SQL, we 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 is 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 examples.

```
--1.1
--TRUE OR UNKNOWN = TRUE
SELECT 1 WHERE ((1=1) OR (NULL=1));

--1.2
--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 equality (=) and inequality (<>) 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

IS NULL | IS NOT NULL

We can experiment with setting the ANSI_NULLS to ON and OFF to review how the behavior of NULL markers change. In the following examples we will set the default ANSI_NULLS setting to ON.

SQL provides two functions for handling NULL markers, IS NULL and IS NOT NULL which we will also demonstrate below.

```
--2.1
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;
```

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.

Now that we have covered some of the basics of NULL markers, lets create some sample data tables and start working through more examples.

SAMPLE DATA

We will use the following tables of fruits and their quantities in our quest to understand the behavior of NULL markers. Using two tables of the same type gives us the best example of understanding NULL markers. We will work with this data throughout these exercises.

The DDL to create these tables have been provided in the GitHub repository located here.

Table A

ID	Fruit	Quantity
1	Apple	17
2	Peach	20
3	Mango	11
4	Mango	15
5	<NULL>	5
6	<NULL>	3

Table B

ID	Fruit	Quantity
1	Apple	17
2	Peach	25
3	Kiwi	20
4	<NULL>	<NULL>

JOIN SYNTAX

NULL markers are neither equal to nor not equal to each other. They are treated as unknowns. This is demonstrated by the below statement, where NUL markers are not present in the result set.

```
--3.1
SELECT DISTINCT
       a.Fruit,
       b.Fruit
FROM   #TableA a INNER JOIN
       #TableB b ON a.Fruit = b.Fruit OR a.Fruit <> b.Fruit;
```

Fruit	Fruit
Apple	Apple
Apple	Kiwi
Apple	Peach
Mango	Apple
Mango	Kiwi
Mango	Peach
Peach	Apple
Peach	Kiwi
Peach	Peach

SEMI AND ANTI JOINS

An **anti-join** uses the NOT IN or NOT EXISTS operators. The **semi join** uses the IN or EXISTS operators.

The benefits of semi and anti-joins are they remove the risk of returning duplicate rows. The result set can only contain the columns from the outer semi-joined table. They are also used for readability, as no fields in the joined table can become part of the queries projection (i.e., the SELECT statement).

Because NULLS are a three valued system (true, false, or unknown), this statement returns an empty dataset as it cannot properly determine if the values in Table A are not in Table B.

```
--4.1
SELECT Fruit
FROM   ##TableA
WHERE  Fruit NOT IN (SELECT Fruit FROM ##TableB);
```

Fruit

<Empty Data Set>

Adding an ISNULL function to the inner query is one way to alleviate the issue of NULL markers.

```
--4.2
SELECT DISTINCT
       Fruit
FROM   ##TableA
WHERE  Fruit NOT IN (SELECT ISNULL(Fruit,'') FROM ##TableB);
```

Fruit
Mango

You can also place predicate logic in the WHERE clause to eradicate the NULL markers.

```
--4.3
SELECT DISTINCT
       Fruit
FROM   ##TableA
WHERE  Fruit NOT IN (SELECT Fruit FROM ##TableB WHERE Fruit IS NOT NULL);
```

Fruit
Mango

The opposite of anti-joins are semi-joins. Using the IN operator, this query will return results and we do not need any special logic to handle NULL markers as we can determine which values are in Table B.

```
--4.4
SELECT DISTINCT
       Fruit
FROM   ##TableA
WHERE  Fruit IN (SELECT Fruit FROM ##TableB);
```

Fruit
Apple
Peach

The IN and NOT IN operators can also take a hard coded list of values as its input. For this example, we use the IN operator. Even though we include a NULL marker in the inner query, the results do not include a NULL marker.

```
--4.5
SELECT Fruit
FROM   ##TableA
WHERE  Fruit IN ('Apple','Kiwi',NULL);
```

Fruit
Apple

EXISTS is much the same as IN. But with EXISTS you must specify a query and you can specify multiple join conditions. A NULL marker is not in the result set as it is undetermined if the NULL marker in Table A is present in Table B.

```
--4.6
SELECT Fruit
FROM   ##TableA a
WHERE   EXISTS (SELECT 1 FROM ##TableB b WHERE a.Fruit = b.Fruit);
```

Fruit
Apple
Peach

Here is the usage of the NOT EXISTS. Same as the EXISTS clause, a NULL marker is returned in the result set as it is undetermined if the NULL marker in Table A is present in Table B.

```
--4.7
SELECT Fruit
FROM   ##TableA a
WHERE  NOT EXISTS (SELECT 1 FROM ##TableB b WHERE a.Fruit = b.Fruit);
```

Fruit
Mango
<NULL>

SET OPERATORS

The SQL standard for SET OPERATORS does not use the term *EQUAL TO* or *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.

The phrase "Null is not distinct from NULL" may seem difficult to understand at first, but this simply means that NULLS are treated as equalities in the context of SET OPERATORS. In the following examples, we see the SET OPERATORS treat the NULL markers differently than the JOIN syntax.

The UNION operator demonstrates that NULL is not distinct from NULL, as the following returns only one NULL marker.

```
--5.1
SELECT DISTINCT Fruit FROM ##TableA
UNION
SELECT DISTINCT Fruit FROM ##TableB;
```

Fruit
<NULL>
Apple
Kiwi
Mango
Peach

The UNION ALL operator returns all values including each NULL marker.

```
--5.2
SELECT DISTINCT Fruit FROM ##TableA
UNION ALL
SELECT DISTINCT Fruit FROM ##TableB;
```

Fruit
<NULL>
Apple
Mango
Peach
<NULL>
Apple
Kiwi
Peach

The EXCEPT operator treats the NULL markers as being not distinct from each other.

```
--5.3
SELECT DISTINCT Fruit FROM ##TableB
EXCEPT
SELECT DISTINCT Fruit FROM ##TableA;
```

Fruit
Kiwi

The INTERSECT returns the following records.

```
--5.4
SELECT DISTINCT Fruit FROM ##TableA
INTERSECT
SELECT DISTINCT Fruit FROM ##TableB;
```

Fruit
<NULL>
Apple
Peach

GROUP BY

The GROUP BY aggregates the NULL markers together.

```
SELECT Fruit,
       COUNT(*) AS Count_Star,
       COUNT(Fruit) AS Count_Fruit
FROM   ##TableA
GROUP BY Fruit;
```

Fruit	Count_Star	Count_Fruit
NULL	2	0
Apple	1	1
Mango	2	2
Peach	1	1

COUNT FUNCTION

The COUNT function removes the NULL markers when a specified field is included in the function but counts the NULL markers when using the asterisk.

```
--7.1
SELECT Fruit,
       COUNT(*) AS Count_Star,
       COUNT(Fruit) AS Count_Fruit
FROM   ##TableA
GROUP BY Fruit;
```

Fruit	Count_Star	Count_Fruit
<NULL>	1	0
Apple	1	1
Mango	1	1
Peach	1	1

CONSTRAINTS

PRIMARY KEYS

The PRIMARY KEY syntax will creae a CLUSTERED INDEX unless specified otherwise. The following statements will error as a PRIMARY KEY does not allow for NULL markers.

```
--8.1
ALTER TABLE ##TableA
ADD CONSTRAINT PK_NULLConstraints PRIMARY KEY NONCLUSTERED (Fruit);

--8.2
ALTER TABLE ##TableA
ADD CONSTRAINT PK_NULLConstraints PRIMARY KEY CLUSTERED (Fruit);
```

A UNIQUE constraint will create a NONCLUSTERED INDEX unless specified otherwise.
The error statement produced will be:
"Cannot define PRIMARY KEY constraint on NULLable column in table '##TableA'."

UNIQUE CONSTRAINTS

To demonstrate that a UNIQUE CONSTRAINT allows only one NULL marker we can run the following statement. We add a UNIQUE CONSTRAINT to Table B, which already includes a NULL marker in the column Fruit.

```
--8.3
ALTER TABLE ##TableB
ADD CONSTRAINT UNIQUE_NULLConstraints UNIQUE (Fruit);
GO

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

The second statement produces the following error.
"Violation of UNIQUE KEY constraint 'UNIQUE_NULLConstraints'. Cannot insert duplicate key in object '##TableB'. The duplicate key value is (<NULL>)."

CHECK CONSTRAINTS

To demonstrate CHECK CONSTRAINTS, let's start by creating a new table with the constraints. The below insert does not error and allows for the operation to take place.

```
--8.4
DROP TABLE IF EXISTS ##CheckConstraints;
GO

CREATE TABLE ##CheckConstraints
(
MyField      INTEGER,
CONSTRAINT Check_NULLConstraints CHECK (MyField > 0)
);
GO

INSERT INTO ##CheckConstraints (MyField) VALUES (NULL);
GO

SELECT * FROM ##CheckConstraints;
```

MyField
<NULL>

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. A PRIMARY KEY constraint does not allow NULL markers, but a UNIQUE constraint allows one NULL marker.

In the below example, we demonstrate that multiple NULL markers can be inserted into a child column that references another column only if the referenced column has a UNIQUE CONSTRAINT on the table. In short, multiple NULL markers can be inserted into the child column.
Also, referential integrity cannot be created on temporary tables, so for this example we create two tables in the dbo schema named Parent and Child.

```
--9.1
DROP TABLE IF EXISTS dbo.Parent;
DROP TABLE IF EXISTS dbo.Child;
GO

CREATE TABLE dbo.Parent
(
ParentID INTEGER PRIMARY KEY
);
GO

CREATE TABLE dbo.Child
(
ChildID INTEGER FOREIGN KEY REFERENCES dbo.Parent (ParentID)
);
GO

INSERT INTO dbo.Parent VALUES (1),(2),(3),(4),(5);
GO

INSERT INTO dbo.Child VALUES (1),(2),(NULL),(NULL);
GO

SELECT * FROM dbo.Parent;
SELECT * FROM dbo.Child;
```

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 attempt to add the value 2 to a NULL marker.

```
--10.1
WITH cte_ColumnExpression AS
(SELECT NULL AS MyInteger
UNION
SELECT 100)
SELECT MyInteger + 2
FROM   cte_ColumnExpression;
```

Fruit	QuantityPlus2
Apple	19
Peach	27
Kiwi	22
<NULL>	<NULL>
<NULL>	<NULL>

SQL FUNCTIONS

Besides the IS NULL and IS NOT NULL predicate logic constructs to query on NULL markers, SQL also provides three functions to help evaluate NULL markers which I demonstrate below.

COALESCE

The COALESCE function returns the first non-null value among its arguments. This function doesn't limit the number of arguments, but they must all be of the same data type.

```
COALESCE(expression [ ,…n ])
```

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)
```

In the examples provided below, note how the result for the ISNULL function is truncated to 'ABC'. The ISNULL function returns the data type of the first argument, which is a VARCHAR(3) in this example. The other difference between ISNULL and COALESCE is that COALESCE can contain multiple parameters and return the parameter that first evaluates to NOT NULL, whereas ISNULL can only take two arguments.

```
--11.1
DECLARE @test VARCHAR(3);
SELECT 'IsNull' AS Type,
       ISNULL(@test, 'ABCD') AS Result
UNION
SELECT 'Coalesce' AS Type,
       COALESCE(@test, 'ABCD') AS Result
UNION
SELECT 'NullIf' AS Type,
       NULLIF('ABCD', @test) AS Result;
```

Type	Result
Coalesce	ABCD
IsNull	ABC
NullIf	ABCD

EMPTY STRING

A useful feature to combat NULL markers in character fields is by using the empty string. The empty string character is not an ASCII value, and the following function returns a NULL marker

```
--12.1
SELECT ASCII('');
```

ASCII
<NULL>

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.

```
--12.2
SELECT DISTINCT
       a.Fruit,
       b.Fruit
FROM   ##TableA a INNER JOIN
       ##TableB b ON ISNULL(a.Fruit,'') = ISNULL(b.Fruit,'');
```

Fruit	Fruit
<NULL>	<NULL>
Apple	Apple
Peach	Peach

CONCAT

The CONCAT function will return an empty string if all the values are NULL.

```
--13.1
SELECT CONCAT(NULL,NULL,NULL) AS ConcatFunc;
```

ConcatFunc

This feature of CONCAT is especially helpful when you want to join to a table on multiple fields (with multiple data types) where NULL markers and empty strings are not used consistently in the tables, as shown below. You can include fields with different data types into the CONCAT function as this function performs an implicit data type conversion.

```
--13.2
WITH
CTE_NULL AS (SELECT 'NULL' AS MyType, NULL AS A, NULL AS B, NULL AS C),
CTE_EmptyString AS (SELECT 'EmptyString' AS MyType, '' AS A, '' AS B, '' AS C)
SELECT *
FROM   CTE_NULL a INNER JOIN
      CTE_EmptyString b ON CONCAT(a.A, a.B, a.C) = CONCAT(b.A, b.B, b.C);
```

MyType	A	B	C	MyType	A	B	C
NULL	<NULL>	<NULL>	<NULL>	EmptyString			

CONCLUSION

Over the course of this document, we have touched on many of the SQL constructs and how they treat NULL markers. I hope this document serves as a guiding document for future development, and always remember to include NULL markers in your test data.

The most important concept to understand with NULL markers is the three-valued logic, where statements can equate to TRUE, FALSE, or UNKNOWN. Understanding the three-valued logic is instrumental in understanding the behavior of NULL markers. TRUE OR UNKNOWN equates to TRUE, and TRUE OR UNKNOWN equates to UNKNOWN.

Other important considerations with NULL markers are the following:

- SQL provides to predicate constructs to evaluate NULL markers, IS NULL and IS NOT NULL.
- Join logic differs from set operators in the context of NULL markers. Joins will treat the NULL markers as neither equal nor not equal to each other, and set operators treat NULL markers as not distinct from each other.
- Semi and anti-joins also have nuances to the how NULL markers are treated. Anti-joins using the NOT IN syntax will return an empty dataset when a NULL marker is present. It is best practice to only use semi and anti-joins on columns that do not contain NULL markers.
- The GROUP BY function will group NULL markers together.
- the COUNT function removes the NULL markers when performed on a specific field but counts the NULL markers when using the COUNT(*).
- Primary keys do not allow a NULL marker, but a UNIQUE constraint allows for one NULL marker in the specified column.
- For referential integrity, multiple NULL markers can be inserted into a child column that references another column.
- When creating a computed column, a NULL marker will return a NULL marker if any type of computation (sum, average, minus, etc.…) is applied.
- SQL Server provides the functions ISNULL, NULLIF, and COALESCE to handle NULL markers.
- Empty strings are often used in place of the NULL marker. When performing data profiling on a dataset, check for empty strings as well as NULL markers. The empty string does not have an ASCII value associated to it.

I hope you have enjoyed this evaluation on the behavior of NULLS. If you have any questions, bug fixes, or want to say hello, please contact me through my website at https://advancedsqlpuzzles.com.

The End!