BEHAVIOR OF NULLS

Scott Peters

https://advancedsqlpuzzles.com

Last updated: 06/28/2022

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 for developers.

To best understand NULL markers, one must understand the three-valued logic of true, false, or unknown, and recognize how 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 from this document can be found in the following Git repository.
https://github.com/smpetersgithub/AdvancedSQLPuzzles/tree/main/Database%20Writings

The examples provided are written in Microsoft's SQL Server T-SQL. The provided SQL statements can be easily modified to fit your dialect of SQL.

I welcome any corrections, new tricks, new techniques, dead links, misspellings, or bugs.

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 ANSI_NULLS session setting 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

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.

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;
```

Now that we have covered the basics of NULL markers, lets create two sample data tables and start working through 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 for 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 NULL 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

Semi-joins and anti-joins are two closely related join methods. The semi-join and anti-join are types of joins between two tables where rows from the outer query are returned based upon the presence or absence of a matching row in the joined table.

Anti-joins use the NOT IN or NOT EXISTS operators. Semi-joins use the IN or EXISTS operators.

There are several benefits of using anti-joins and semi-joins over INNER joins:

1. Semi-joins and anti-joins remove the risk of returning duplicate rows.
2. Semi-joins and anti-joins increase readability as the result set can only contain the columns from the outer semi-joined table.

There are several key differences between semi-joins and anti-joins:

1. The NOT IN operator will return an empty set if the anti-join contains a NULL marker. The NOT EXISTS will return a dataset that contains a NULL marker if the anti-join contains a NULL marker.
2. The IN and EXIST operators will return a dataset if the semi-join contains a NULL marker.
3. The NOT EXISTS and EXIST operators can join on multiple columns between the outer and inner SQL statements. The NOT IN or IN operators join on only one single field.

If you are performing an anti-join to a NULLable column, consider using the NOT EXISTS operator over the NOT operator.

This statement returns an empty dataset as the anti-join contains a NULL marker.

```
--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

A better practice is to 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 a result set. A NOT IN operator would return an empty dataset.

```
--4.4
SELECT 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.

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

Fruit
Apple

Here is the usage of the NOT EXISTS. A NULL marker is returned in the dataset (unlike the IN operator).

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

Fruit
Peach
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.

```
--6.1
SELECT Fruit,
       COUNT(*) AS Count_Star,
       COUNT(Fruit) AS Count_Fruit
FROM   ##TableA
```

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

COUNT AND AVERAGE 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

The AVG function will remove records with NULL markers in its calculation, as shown below. For this example, we created the test data in the common table expression `cte_Average`, as this more easily demonstrates the functions behavior.

In SQL Server, when performing division between integers, you will need to cast or convert the values to decimal before division, as shown below.

```
--7.2
WITH cte_Average AS
(
SELECT 1 AS Id, 100 AS MyValue
UNION
SELECT 2 AS Id, 100 AS MyValue
UNION
SELECT 3 AS Id, NULL AS MyValue
UNION
SELECT NULL AS Id, 100 AS MyValue
)
SELECT SUM(MyValue) / CAST(COUNT(*) AS NUMERIC(10,2)) AS Average_CountStar,
       SUM(MyValue) / CAST(COUNT(Id) AS NUMERIC(5,2)) AS Average_CountId,
       AVG(CAST(MyValue AS NUMERIC(10,2))) AS Average_AvgFunction
FROM   cte_Average;
```

Average_CountStar	Average_CountId	Average_AvgFunction
112.50000000000	150.000000	150.000000

CONSTRAINTS

PRIMARY KEYS

The PRIMARY KEY syntax will create 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 short, multiple NULL markers can be inserted into the child column or a FOREIGN KEY constraint.

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.

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.Child;
DROP TABLE IF EXISTS dbo.Parent;
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 the Quantity field which includes a NULL marker in our test data.

```
--10.1
SELECT Fruit,
       Quantity + 2 AS QuantityPlus2
FROM   ##TableB;
```

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

When creating a table with a non-NULLable computed column, you must create the column as PERSISTED, else you will receive the error message:

```
Msg 8183, Level 16, State 1, Line 307
Only UNIQUE or PRIMARY KEY constraints can be created on computed columns, while CHECK,
FOREIGN KEY, and NOT NULL constraints require that computed columns be persisted.
```

From this error message we can see there are several rules we need to follow on computed columns if we want to add UNIQUE, PRIMARY KEY, FOREIGN KEY, CHECK and NOT NULL constraints.

Here we add a NOT NULL constraint to a PERSISTED computed column and add a PRIMARY KEY to the column.

```
--10.2
DROP TABLE IF EXISTS MyComputed;
GO

CREATE TABLE MyComputed
(
Int1 INTEGER NOT NULL,
Int2 INTEGER NOT NULL,
Int3 AS MyInt1 + MyInt2 PERSISTED NOT NULL
);

ALTER TABLE MyComputed ADD PRIMARY KEY CLUSTERED (Int3);

DROP TABLE IF EXISTS MyComputed;
GO
```

SQL FUNCTIONS

Besides the IS NULL and IS NOT NULL predicate logic constructs, SQL also provides three functions to help evaluate NULL markers.

COALESCE

The COALESCE function returns the first non-NULL value among its arguments. If all values are NULL, COALESCE returns a NULL marker.

```
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)
```

COALESCE and ISNULL have several key differences that should be noted. A full comparison can be found here in this Microsoft documentation.

The major differences between COALESCE and ISNULL from the documentation are:

1) COALESCE behaves the same as a CASE statement, and therefore can accept multiple parameters and return a NULL marker. ISNULL cannot return a NULL marker and accepts only two parameters.
2) COALESCE determines the type of output based upon data type precedence. ISNULL uses the data type of the first parameter.

Check the Microsoft documentation provided above for more robust examples of the differences between COALESCE and ISNULL. Below I provide a few quick examples to note their behavior.

```
--11.1
WITH cte_functions AS
(
SELECT  1 AS Id,
        'IsNull' AS Type,
        ISNULL(@test, 'ABCD') AS Result--truncates ABCD to ABC.
UNION
SELECT  2 AS Id,
        'Coalesce_3Parameters' AS Type,
        COALESCE(@test, 'ABCD', 'EFGH') AS Result1--accepts three parameters, does
not truncate ABCD
UNION
SELECT  3 AS Id,
        'Coalesce_NULL' AS Type,
        COALESCE(@test, NULL, NULL) AS Result1--Returns a NULL
UNION
SELECT  4 AS Id,
       'NullIf' AS Type,
        NULLIF('ABCD', @test) AS Result--The first argument needs to be a NON-NULL
)
SELECT   *
FROM     cte_functions
```

Type	Type	Result
1	IsNull	ABC
2	Coalesce_3Parameters	ABCD
3	Coalesce_NULL	<NULL>
4	NullIf	ABCD

EMPTY STRINGS, NULL, AND ASCII VALUES

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. Also, you would assume the ASCII value for a NULL marker is 0 when reviewing an ASCII code chart, however this is not the case, and the ASCII function returns NULL for the NULL marker as shown below. SQL does not use the standard ANSI NULL marker.

```
--12.1
SELECT ASCII('') AS ASCII_EmptyString,
       ASCII(NULL) AS ASCII_NULL;
```

ASCII_EmptyString	ASCII_NULL
<NULL>	<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			

VIEWS

When creating a view, if you perform a CAST function or create a computed column on a column which has a NOT NULL constraint, the result will yield a NULLable column.

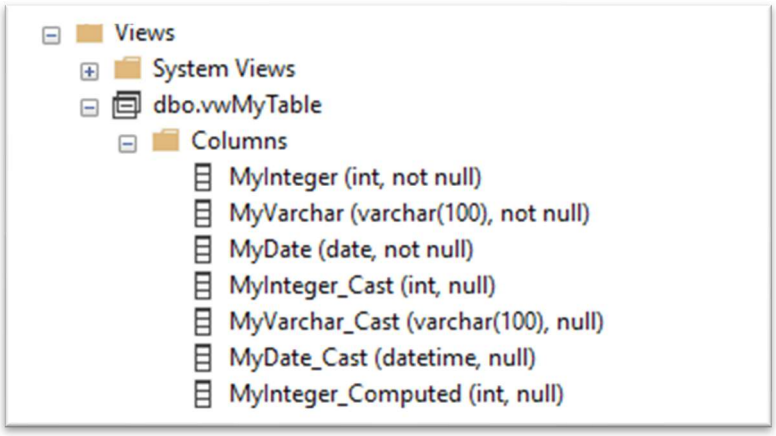
Here we create a table with NOT NULL constraints and then create a view where we perform a CAST and create a computed column. From the resulting screen shot, we can see the columns are NULLable.

```
--14.1
DROP TABLE IF EXISTS MyTable;
GO

CREATE TABLE MyTable
(
MyInteger INT NOT NULL,
MyVarchar Varchar(100) NOT NULL,
MyDate Date NOT NULL
);
GO

CREATE OR ALTER VIEW vwMyTable AS
SELECT MyInteger,
       MyVarchar,
       MyDate,
       CAST(MyInteger AS INT) AS MyInteger_Cast,
       CAST(MyVarchar AS VARCHAR(100)) AS MyVarchar_Cast,
       CAST(MyDate AS DATETIME) AS MyDate_Cast,
       MyInteger * 10 AS MyInteger_Computed
FROM   MyTable;
GO
```

Here is a screen shot of the resulting view. The columns with CAST and COMPUTE are NULLable.



BOOLEAN VALUES

Here we will discuss two SQL constructs, the BIT data type and the NOT operator.

BIT

Often, we think of the BIT data type as being a Boolean value (true or false, yes or no, on or off, one or zero…), however NULL markers are allowed for the BIT data type making the possible values 1, 0 and NULL. In short, the BIT data type in SQL is not a true Boolean value.

Because the only acceptable values for the BIT data type is 1, 0 or NULL. The BIT data type converts any nonzero value to 1. As discussed earlier, the NULL marker is neither a nonzero value nor a zero value, so it is not promoted to the value of 1. Here we can demonstrate that behavior.

```
--15.1
SELECT CAST(NULL AS BIT)
UNION
SELECT CAST(3 AS BIT);
```

MyBit
<NULL>
1

NOT

The NOT operator negates a Boolean input.

Here we can see how the NOT operator acts on our test data. The distinct fruits in the table are Apple, Peach, Mango and NULL. Placing the NOT operator on the predicate logic will return Apple and Peach, but the NULL marker is not returned.

```
--15.2
SELECT  *
FROM    ##TableA
WHERE   NOT(FRUIT = 'Mango');
```

ID	Fruit	Quantity
1	Apple	17
2	Peach	20

RETURN

The RETURN statement exists unconditionally from a query or procedure. All stored procedures return a value of 0 for a successful execution, and a nonzero value for a failure. When the RETURN statement is used with a stored procedure, it cannot return a NULL marker. If a procedure tries to return a NULL marker in the RETURN statement, a warning message is generated and a value of 0 is returned.

Here we will create a stored procedure that overrides the default RETURN value and attempt to return a NULL value.

```
--16.1
CREATE OR ALTER PROCEDURE SpReturnStatement
AS
IF 1=2
    RETURN 1
ELSE
    RETURN NULL;
GO

DECLARE @return_status INT;

EXEC @return_status = SpReturnStatement;
SELECT 'Return Status' = @return_status;
```

```
The 'SpReturnStatement' procedure attempted to return a status of NULL, which is not
allowed. A status of 0 will be returned instead.
```

Return Status
0

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 most importantly, 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.

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