2021-05-14T17:00:00+02:00
James Wondrasek

How to Use Comparison Operators with NULLs in SQL

The SQL NULL value serves a special purpose. It also comes with counterintuitive behaviors that can trip up SQL beginners or even experienced programmers. Learn how to avoid these problems when you use NULL with comparison operators.

This article is going to help you master best practices for crafting SQL queries that work with NULL values and use comparison operators (=, <>, <, >) – which, if you have written any SQL queries before, you know is just about every query you will ever write! This is essential knowledge and mastering it will make SQL easier for you.

We're going to take a quick tour of SQL NULLs, why they exist, and how their weirdness impacts query results. This will be followed by some SQL queries that demonstrate the weirdness. Then we'll discuss standard techniques for writing queries that correctly deal with NULLs and comparison operators. Finally, there will be a quick summary of all SQL's comparison operators and how each interacts with NULL.

Do you already know SQL but don't know what to do with it? Check out our interactive SQL Practice Set course!

Don't worry. It's not as hard as it sounds. Let's get started by reviewing NULL values in SQL.

SQL NULL — The Known Unknown

Databases are meant to be a single source of truth. What is recorded in the fields for each row represents what is known.

Look at the table below, which has been adapted from What Is a NULL in SQL (a great resource for a deep dive into SQL NULLs). This could be part of a database created by a *Simpsons* superfan.

name	social_sec_no	status	spouse
Homer Simpson	000-00-5000	married	Marjorie Bouvier
Nedward Flanders	000-00-4000	married	Maude Flanders
Waylon Smithers	000-00-8000	single	NULL
Dr Nick Riviera	000-00-7000	NULL	NULL

NULL is representing two different things here. Waylon Smithers is known to be single, so NULL in this row in the spouse column represents a *non-existent value*. But we know so little about Dr. Nick that the NULL values in his spouse and `status` columns represent an *unknown value*.

To maintain the integrity of a database, both interpretations of NULL are necessary. And to help programmers maintain that integrity despite missing and unknown values, SQL incorporates NULLs into its ternary logic.

What Is Ternary Logic and How Does It Work in SQL?

Binary logic uses two values: True and False, 0 and 1, etc. Ternary logic uses three values. In SQL, those three values are True, False and Unknown. In SQL ternary logic, NULL equates to the value of unknown.

Here is how the three values of ternary logic operate with SQL's logical `NOT`, `OR` and `AND` operators:

NOT	
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

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

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN

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

The take-away: **Any logical operation involving a NULL results in a value of *unknown* except for TRUE OR NULL and FALSE AND NULL**. Interesting, but not something you need to memorize. As you will see, SQL helps us work around this tricky logic. Which is handy, because SQL comparison operations use the same ternary logic.

SQL Comparison Operations and NULL Values

Here is a query using comparison and logical operators. Guess how many rows it returns:

```
SELECT spouse
  FROM simpsons
 WHERE    spouse = NULL
    OR NOT (spouse = NULL)
```

Let's start with the first comparison operation:

```
WHERE    spouse = NULL
```

Whatever the comparison column contains – salaries, pet names, etc. – if we test that it is equal to NULL, the result is **unknown**. This is true even if the column value is NULL. This is what confuses programmers who are experienced in other languages. For example, in Python the *None* value seems similar to SQL NULL and can be compared against itself:

```
>>> None == None
True
```

But the first line of the `WHERE` clause is going to return unknown. So as our query is evaluated, it is going to look like this:

```
SELECT spouse
  FROM simpsons
 WHERE unknown
    OR NOT (spouse = NULL)
```

Let's look at the second line of the WHERE condition:

```
OR NOT (spouse = NULL)
```

This comparison is also going to return *unknown*. We can see from the truth table earlier that *NOT unknown* is going to return *unknown*. So now our query has become:

```
SELECT spouse
  FROM simpsons
 WHERE unknown
    OR unknown
```

The OR truth table above tells us the result from this will be *unknown*.

A WHERE clause requires **true** conditions. The result *not* being false is not enough. So despite the query looking like it will return every row, the ternary logic of SQL and the nature of SQL NULL results in **zero rows being returned**.

SQL's Other Comparison Operators

For these examples, we are going to use a different table, `pet_owners`:

name	pet_count
Bob	5
Cate	2
Alice	NULL
Steve	22

NULL and the < Operator

Using the `pet_owners` table, let's see who has less than 5 pets.

The result:

name
Cate

Why? Alice, who hasn't completed her pet survey form, has NULL for her `pet_count`. The value of any NULL is unknown. Is NULL < 5? That's unknown, so Alice cannot be included in the results.

NULL and the > Operator

Now we will see who has more than 3 pets.

```
SELECT p.name
  FROM pet_owners p
 WHERE pet_count > 3
```

The result:

name
Bob
Steve

Why? Again it is the unknown value of NULL. Just as it is unknown if NULL < 5, it is unknown if NULL > 3. Alice is excluded from the results.

NULL and the <= Operator

In a subtle change, we will now get a list of everyone with no more than 5 pets.

```
SELECT p.name
  FROM pet_owners p
 WHERE pet_count <= 5
```

The result:

name
Bob
Cate

Changing our first query from using < to using <= adds Bob to the result set, but not Alice. In the first query, Bob's pet count (5) is not *less than* 5. But it is less than or equal to 5, so he is now included in the query result. Alice still doesn't appear.

When examining Alice's row, we can think of <= as shorthand for "NULL < 5 OR NULL = 5". We know from above that "NULL < ANYTHING" will return *unknown*; from earlier, we know that "NULL =

ANYTHING" will also return *unknown*. Once again, Alice is excluded from the results.

NULL and the >= Operator

Now let's see who has at least 3 pets.

```
SELECT p.name
  FROM pet_owners p
 WHERE pet_count >= 3
```

The result:

name
Bob
Steve

Like <=, we can think of >= as a logical combination of two comparison operations. So, for Alice, the comparison is equivalent to "NULL > 3 OR NULL = 3". You should now be clear that this can only result in a value of *unknown*.

SQL Comparison Operators that Work with NULLs

To handle NULLs correctly, SQL provides two special comparison operators: IS NULL and IS NOT NULL. They return only *true* or *false* and are the best practice for incorporating NULL values into your queries.

To get the result we expected, we can rewrite the Simpsons query like so:

```
SELECT spouse
  FROM simpsons
 WHERE spouse IS NULL
    OR spouse IS NOT NULL
```

Now the query will return every row, as we expected.

SQL NULL and Outer Joins

This is a more advanced topic. If you are new to SQL JOINs, you should probably read 7 SQL JOIN Examples with Detailed Explanations first.

It is common to use a WHERE clause along with the `IS NOT NULL` operator to get rid of rows with NULL values. But this can sometimes be an inefficient way of achieving a result. Here's why.

Outer joins — a `LEFT`, `RIGHT`, or `FULL JOIN` — can be thought of as an `INNER JOIN` (which returns matched rows) plus unmatched rows with columns filled in with NULLs.

A `LEFT JOIN` returns all the rows in the left table of the join with the matching rows from the right table (or NULL values where there isn't a match). A `RIGHT JOIN` returns all the rows from the right table with matching rows (or NULLs) from the left table. A `FULL JOIN` is like an `INNER JOIN` that also returns all the unmatched left and right table rows extended by NULLs.

If your query's `WHERE` clause is filtering out rows that have been extended by NULLs, you are basically canceling your outer join. You should rewrite your query using an `INNER JOIN`.

In other cases, NULLs will interact with your WHERE clause and cause incorrect results. Let's look at an example that demonstrates this. You can run it yourself using this SQL Fiddle. (Here's a guide to more sites to use for SQL practice.)

For this example, we're going to use two tables, one for **`customers`** and one for **`orders`**.

id	name	email
1	Alice	alice@gmail.com
2	Bob	bob@hmail.com
3	Cate	cate@imail.com

Table **`customers`**

id	order_date	cust_email	amount
1	2021-02-04	bob@hmail.com	50
2	2021-02-05	cate@imail.com	20

id	order_date	cust_email	amount
3	2021-02-06	cate@imail.com	40
4	2021-02-06	bob@hmail.com	15

Table `orders`

We want to see the amount spent by all of our registered customers since February 4, 2021 ("2021-02-04"). We begin with an INNER JOIN. There are a few extra SQL functions in this query that might be new to you, but LearnSQL has you covered. We have articles on COALESCE, aggregate functions like SUM, and the GROUP BY clause. You don't need to worry about what these functions do right now. Just focus on what each query returns, starting with this one:

```
SELECT c.name, COALESCE(SUM(o.amount),0) as 'Total'
FROM customers c
INNER JOIN orders o
ON c.email = o.cust_email
WHERE o.order_date > '2021-02-04'
GROUP BY c.name;
```

The query outputs this result:

name	Total
Bob	15
Cate	60

It looks good, but we have 3 customers. If we want to see every customer, we need to use a `LEFT OUTER JOIN` (aka `LEFT JOIN` for short). It will include every row in the left table of the `FROM` statement even if there is no matching data in the right table. This gives us our next query:

```
SELECT c.name, COALESCE(SUM(o.amount),0) as 'Total'
FROM customers c
LEFT JOIN orders o
ON c.email = o.cust_email
WHERE o.order_date > '2021-02-04'
GROUP BY c.name;
```

The results might surprise you:

name	Total
Bob	15
Cate	60

Why is this happening? Why is Alice still missing? A simpler query will give us a hint:

```
SELECT c.name, COALESCE(SUM(o.amount),0) as 'Total'
FROM customers c
LEFT JOIN orders o
ON c.email = o.cust_email
WHERE o.order_date > '2021-02-04'
GROUP BY c.name;
```

The result:

name	order_date
Bob	2021-02-04
Cate	2021-02-05
Cate	2021-02-06
Bob	2021-02-06
Alice	(null)

As stated earlier, `LEFT JOIN` includes all rows from the left table. Where there is no matching row in the right table, the columns are filled with NULLs.

The WHERE clause performs its filtering *after* `JOIN`, so all rows for Alice will be removed because comparing anything to NULL, such as a non-existent `order_date`, returns *unknown.* `WHERE` only returns a row where the conditionals evaluate to TRUE.

The way to fix this is to move the conditional expression, here `o.order_date > '2021-02-04'`, into the `JOIN` by including it in the `ON` clause:

```
SELECT c.name, COALESCE(SUM(o.amount),0) as 'Total'
FROM customers c
LEFT JOIN orders o
ON c.email = o.cust_email
WHERE o.order_date > '2021-02-04'
GROUP BY c.name;
```

Because we are using a `LEFT JOIN`, Alice remains in the result set despite the additional date condition. Her NULL is turned into a cleaner "0" by the `COALESCE()` function. The query result is now what we were expecting:

name	Total
Alice	0
Bob	15
Cate	60

Improve your SQL skills with our SQL Practice Set: 88 hands-on exercises that you can solve in your browser!

Learn More about SQL NULLs

You should now understand how SQL treats NULL values and the best practices for working with them. You know about `IS NULL` and `IS NOT NULL` operators and how SQL's ternary logic always returns *unknown* when anything is compared to a NULL *except for two special cases*. You've also seen how to rewrite queries so you don't need to filter out those troublesome NULLs.

But there's more to know about working with NULLs in SQL. I suggest you continue learning with the articles How ORDER BY and NULL Work Together in SQL and NULL Values and the GROUP BY Clause.

And if you want to truly master SQL, I recommend the SQL Practice Set or the SQL Practice track. They offer a thorough and easy-to-follow learning experience that will help you hone your craft.
