SQL Antipatterns, Volume 1

Avoiding the Pitfalls of Database Programming

The Pragmatic Bookshelf
Raleigh, North Carolina

The Pragmatic Programmers

SQL Antipatterns, Volume 1

Avoiding the Pitfalls of Database Programming

Bill Karwin

edited by Jacquelyn Carter

SQL Antipatterns, Volume 1

Avoiding the Pitfalls of Database Programming

Bill Karwin

The Pragmatic Bookshelf
Raleigh, North Carolina

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Jacquelyn Carter
Copy Editor: Karen Galle
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

ISBN-13: 978-1-68050-898-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—November 2022

To my wife Jan, my best supporter.

Solution: Use Null as a Unique Value

Most problems with null values are based on a common misunderstanding of the behavior of SQL's three-valued logic. For programmers accustomed to the conventional true/false logic implemented in most other languages, this can be a challenge. You can handle null values in SQL queries with a little study of how they work.

Null in Scalar Expressions

Suppose Stan is thirty years old, while Oliver's age is unknown. If you ask whether Stan is older than Oliver, the only possible answer is "I don't know." If you ask whether Stan is the same age as Oliver, the answer is also "I don't know." If you ask what is the sum of Stan's age and Oliver's age, the answer is the same.

Charlie's age is also unknown. If you ask whether Oliver's age is equal to Charlie's age, the answer is still "I don't know." This shows why the result of a comparison like `NULL = NULL` is also null.

The following table describes some cases where programmers expect one result but get something different.

Expression	Expected	Actual	Because
NULL = 0	TRUE	NULL	Null is not zero.
NULL = 12345	FALSE	NULL	Unknown if the unspecified value is equal to a given value.
NULL <> 12345	TRUE	NULL	Also unknown if it's unequal.
NULL + 12345	12345	NULL	Null is not zero.
NULL || 'string'	'string'	NULL	Null is not an empty string.
NULL = NULL	TRUE	NULL	Unknown if one unspecified value is the same as another.
NULL <> NULL	FALSE	NULL	Also unknown if they're different.

Of course, these examples apply not only when using the `NULL` keyword but also to any column or expression whose value is null.

Null in Boolean Expressions

Null is neither true nor false. A null value certainly isn't true, but it isn't the same as false. If it were, then applying `NOT` to a null value would result in true. However, that's not the way it works; `NOT (NULL)` results in another null. This confuses some people who try to use boolean expressions with null.

The following table shows some some additional cases where programmers expect one result but get something different.

Expression	Expected	Actual	Because
NULL AND TRUE	FALSE	NULL	Null is not false.
NULL AND FALSE	FALSE	FALSE	Any truth value AND FALSE is false.
NULL OR FALSE	FALSE	NULL	Null is not false.
NULL OR TRUE	TRUE	TRUE	Any truth value OR TRUE is true.
NOT (NULL)	TRUE	NULL	Null is not false.

The Right Result for the Wrong Reason

Consider the following case, where a nullable column may behave in a more intuitive way by serendipity.

```
SELECT * FROM Bugs WHERE assigned_to <> 'NULL';
```

Here the nullable column assigned_to is compared to the string value 'NULL' (notice the quotes), instead of the actual NULL keyword.

Where assigned_to is null, comparing it to the string 'NULL' is not true. The row is excluded from the query result, which is the programmer's intent.

The other case is that the column is an integer compared to the string 'NULL'. The integer value of a string like 'NULL' is zero in most brands of database. The integer value of assigned_to is almost certainly greater than zero. It's unequal to the string, so the row is included in the query result.

Thus, by making another common mistake, that of putting quotes around the NULL keyword, some programmers may unwittingly get the result they wanted. Unfortunately, this coincidence doesn't hold in other searches, such as WHERE assigned_to = 'NULL'.

Searching for Null

Since neither equality nor inequality return true when comparing one value to a null value, you need some other operation if you are searching for a null. Older SQL standards define the IS NULL predicate, which returns true if its single operand is null. The opposite, IS NOT NULL, returns false if its operand is null.

Fear-Unknown/soln/search.sql

```
SELECT * FROM Bugs WHERE assigned_to IS NULL;

SELECT * FROM Bugs WHERE assigned_to IS NOT NULL;
```

In addition, the SQL-99 standard defines another comparison predicate, IS DISTINCT FROM. This works like an ordinary inequality operator <>, except that it always returns true or false, even when its operands are null. This relieves you from writing tedious expressions that must test IS NULL before comparing to a value. The following two queries are equivalent:

Fear-Unknown/soln/is-distinct-from.sql
```
SELECT * FROM Bugs WHERE assigned_to IS NULL OR assigned_to <> 1;

SELECT * FROM Bugs WHERE assigned_to IS DISTINCT FROM 1;
```

You can use this predicate with query parameters to which you want to send either a literal value or NULL:

Fear-Unknown/soln/is-distinct-from-parameter.sql
```
SELECT * FROM Bugs WHERE assigned_to IS DISTINCT FROM ?;
```

Support for IS DISTINCT FROM is inconsistent among database brands. PostgreSQL, IBM DB2, and Firebird do support it, whereas Oracle and Microsoft SQL Server don't support it yet. MySQL offers a proprietary operator <=> that works like IS NOT DISTINCT FROM.

Declare Columns NOT NULL

It's recommended to declare a NOT NULL constraint on a column for which a null would break a policy in your application or otherwise be nonsensical. It's better to allow the database to enforce constraints uniformly rather than rely on application code.

For example, it's reasonable that any entry in the Bugs table should have a non-null value for the date_reported, reported_by, and status columns. Likewise, rows in child tables like Comments must include a non-null bug_id, referencing an existing bug. You should declare these columns with the NOT NULL option.

Some people recommend that you define a DEFAULT for every column, so that if you omit the column in an INSERT statement, the column gets some value instead of null. That's good advice for some columns but not for other columns. For example, Bugs.reported_by should not be null. It should be the account id of the user who reported it, but it's not possible to declare this as a default. It's valid and common for a column to need a NOT NULL constraint yet have no logical default value.

Dynamic Defaults

In some query results, you may need to force a column or expression to be non-null for the sake of simplifying the query logic, but you don't want that value to be stored in the table. You need a way to set a non-null value to be used if a given expression would return a null result. For this you should use the COALESCE() function. This function accepts a variable number of arguments and returns its first non-null argument.

In the story about concatenating users' names in the opening of this chapter, you could use COALESCE() to make an expression that uses a single space in place of the middle initial, so a null-valued middle initial doesn't make the whole expression become null.

Fear-Unknown/soln/coalesce.sql

```sql
SELECT first_name || COALESCE(' ' || middle_initial || ' ', ' ') || last_name
  AS full_name
FROM Accounts;
```

COALESCE() is a standard SQL function. Some database brands support a similar function by another name, such as NVL() or ISNULL().

Use null to signify a missing value for any data type.

Mini-Antipattern: NOT IN (NULL)

If the logic of null isn't confusing enough, there are edge cases where it's even harder to avoid getting lost in the boolean rules.

You may have mastered the logic enough to understand that the following two queries are equivalent:

Fear-Unknown/mini/in-null.sql

```sql
SELECT * FROM Bugs WHERE status IN (NULL, 'NEW');

SELECT * FROM Bugs WHERE status = NULL OR status = 'NEW';
```

You know that comparing a value equals null is unknown, and that's not true, so the first term of that comparison will never be satisfied. That's okay, because the query still matches rows with "NEW".

This gets really interesting when the search is negated.

Fear-Unknown/mini/not-in-null.sql

```sql
SELECT * FROM Bugs WHERE status NOT IN (NULL, 'NEW');
```

You might think this simply matches the complement of the set of rows matched by the previous query. That is, all rows except those with status "NEW". In fact, *none* of the rows match. Why?

The query with the NOT IN predicate can be rewritten as either of the following:

Fear-Unknown/mini/not-in-null.sql
```
SELECT * FROM Bugs WHERE NOT (status = NULL OR status = 'NEW');

SELECT * FROM Bugs WHERE NOT (status = NULL) AND NOT (status = 'NEW');
```

The first rewrite looks familiar, as an IN predicate is equivalent to equality comparisons to each respective value, as terms of OR operations. Then the negation NOT is applied to the expression. You know by now that comparing a column equal to null is unknown, and the negation of unknown is still unknown.

The second rewrite is an application of *DeMorgan's law*, a boolean algebra transformation. The negation of an expression negates each term in the expression, as it converts OR to AND or vice versa.

Now you should see that NOT (status = NULL) will still be unknown, and using AND to combine that with the other term makes the whole expression unknown for any row evaluated. So, the SQL query always fails to match any rows, regardless of any value in the status column.