Database Design Follies: NULL vs. NOT NULL

By Edward Pollack, 2016/02/26 (first published: 2014/10/30)

Introduction

When designing a new database, one of the first decisions we make it to define our columns. This process involves data types, constraints, indexes, keys, and to ensure that our definition accurately reflects the data that the column represents. Deciding if a column should be NULL or NOT NULL will be an important piece of this work, and sometimes far more contentious than we'd expect of a decision that has only 2 possible results!

Background

In the world of SQL, NULL occupies a special place. It is not a value, nor is it a state such as true or false, but it's an absence of any known value. NULL is used explicitly to indicate when a value is unknown, undefined, does not, or cannot exist.

This has some big implications for the future of our fledgling database. Since NULL isn't a value like 17 or 'Purple', it doesn't behave the same way when used in arithmetic, string manipulation, aggregation, or conditionals. What exactly does 42 + NULL equal? How about 'Customer revenue was ' + NULL?

The short answer is NULL, the longer answer is to never allow that situation to arise in the first place because it's as confusing as dividing by zero. In fact, it's worse as division by zero will typically throw an error, whereas multiplying by NULL will trickle through your SQL into the application and only cause trouble when NULL is no longer a viable result.

NULL may now sound like a bad idea---but in fact it has many excellent uses. We do run into many scenarios where an account doesn't have a contract, a work order has yet to be assigned a complete date, or you don't have a favorite color and elected to not fill in the optional field on the web site.

There are DBAs that vigilantly contend that all columns should be NOT NULL, whereas others will allow NULL wherever they can. This is not a debate that is worth the space here, but instead we'll look at the pros and cons of NULL-able columns, when they are best used, and some common mistakes. Getting these decisions right when you first design new tables will save immense trouble down the road. Changing this after years of SQL and code have been built on top of it could easily be an exercise in tedium (at best), or implementing cheap hacks (at worst).

For the remainder of this article, we'll use the following simple example to illustrate NULL usage:

```
CREATE TABLE Accounts
(
```

```
AccountID INT IDENTITY(1,1) NOT NULL
AccountName VARCHAR(100) NOT NULL,
Address_1 VARCHAR(50) NOT NULL,
Address_2 VARCHAR(50) NULL, -- Optio
City VARCHAR(35) NOT NULL,
StateOrTerritory VARCHAR(25) NOT NUL
Country VARCHAR(25) NOT NULL,
ContractID INT NULL, -- FK to Contra
CurrencySymbol VARCHAR(1) NOT NULL,
AccountIcon VARCHAR(50) NOT NULLCONS
-- The name of the icon that will di
CONSTRAINT PK_Accounts PRIMARY KEY C
```

A final note: All information provided here will assume that ANSI_NULLS is ON. Additional information will be provided for select cases when ANSI_NULLS is OFF, as needed. Microsoft has deprecated ANSI_NULLs and in a future version of SQL Server will make ON the required default for this setting. At that time, any SQL that sets ANSI_NULLS OFF will generate an error. As you write new code, be sure to take this into account and not rely on the old OFF setting for this SET statement. In addition, SET ANSI_NULLS OFF does not follow the ISO standard, which means that SQL written with this setting will behave unreliably in other SQL systems.

Guidelines

The first key to using NULLs is to define what NULL means for any given column. This should be included in your design documentation so that DBAs and developers that refer to your table in the future will have no trouble understanding what a column means, what sort of data it represents, and what a NULL means, if allowed.

OPTIONAL OR REQUIRED?

The first and easiest criteria to determine if a column should be NULL or not is if it is optional. In the web form for your application, what fields will throw errors back at the user if they choose to leave them blank? What data is necessary to properly define an entity? These can always be NOT NULL columns as the application will guarantee a value. In addition, the NOT NULL ensures that anyone manually manipulating data in the table cannot inadvertently place a NULL in this column and break your application. In the above example, Address_2 allows NULL values because many people don't have a second line to their address and would want to leave the field blank. Optional foreign keys, such as ContractID should also be NULL, as an account with no contract cannot populate that column with any meaningful data.

On the other hand, AccountName and Address_1 are NOT NULL as they would be required by any application that uses this table. They should never be undefined and if we were to allow NULLs in these columns, we would be forced to ask many questions. Is this data useful? Will it break the application? How is it possible for a customer to not have a name? The answers to these questions would quickly lead us to determine that there should not and cannot be NULL values in these columns.

DATA THAT IS CREATED IN THE FUTURE

Some columns contain data that is always known. For example, your country will always exist from the moment an account is created until the moment it is deleted. There is never any point in time in which we cannot or do not know the country in which an account is located. If we one day have an account located on the moon, we'd simply populate Country with something new to indicate its lunar location. Other times, we need to store values that are not present initially in our newly inserted rows, but will be (or may be) populated later. Resolutions, completion dates, status messages,

and so on are all examples of columns that will likely start off NULL, but eventually be given a value as time goes on.

NORMALIZATION

It is possible to remove most instances of NULL columns through further normalization. Data that has to do with optional keys, completion dates, or status/error messages can be offloaded to separate tables and keyed back to the parent table. For example, we can remove the Address_2 column into another table with any other optional descriptive elements and foreign key them back to the Accounts table. This would allow us to only populate rows in the new table if and when we had an account with an Address_2, but we would then also need to support a new table as well as some constraint to ensure that an account can only ever have a single Address_2, and that one-to-many relationships are not allowed.

Is this worth it? I tend to look at tables as reflections of an object, and when we query the table, we are looking to return some set of data describing that object. If we always need to check for a status message or completion data, then we are forcing extra joins and existence checks into our business logic to support the separated parts of our object. If the additional columns are only needed on special occasions, or make up a separate aspect of an account, then a separate table may be sensible.

SELF-DOCUMENTATION

Good tables are (practically) self-documenting. The column names, data types, foreign key relationships, and other elements of their structure will tell you what they mean, how they are joined, and how to effectively use them to retrieve the data you're looking for. The rationale behind a column's nullability should be self-evident. If we need to frequently ask what a NULL means, why a column can allow NULLs, why a column doesn't allow NULLs, or what a blank, -1, or other dummy value means, then we need to review our table and ask if we didn't goof while designing it. If the documentation doesn't affirm the structure of a table, then that disconnect may be a reason for investigating potential changes to either.

Oftentimes the best indicator of bad database design is the frequent workarounds that are silently constructed to prevent any fundamental changes to our existing system. If we allow NULL in a StatusID column, but then apply ISNULL(StatusID, 17) whenever we use the column, it is worth asking if we shouldn't have made the column NOT NULL in the first place and forced a default from the application or a default constraint. If we have a NOT NULL INT column where we consistently populate -1 for unknown values, we again should stop and ask why we are going through all of this extra trouble.

There is no 100% right or wrong answer that fits all situations. The answer lies in how a table will be queried against and the most common use-cases for the columns that could be NULL.

NULL Behavior

NULL does not behave in the same way that standard values do. Some behavior is expected and we tend to overlook, but other results can break applications or provide erroneous data if not accounted for.

ARITHMETIC

Any string manipulation, arithmetic, dynamic SQL building, or math involving a NULL value will return a NULL.

```
SELECT 17 + 42 + 289 + NULL
```

This will return NULL.

```
DECLARE @CMD VARCHAR(MAX)
```

```
SELECT @CMD = 'This is a test of how NULL '
SELECT (@CMD)
```

This will also return NULL.

WHERE CLAUSE

Any WHERE clause that checks if column_name = NULL will return no rows, such as in this example. Always use WHERE column_name IS NOT NULL or WHERE column_name IS NULL instead.

```
SELECT AccountID FROM dbo.Accounts WHERE Co
```

With ANSI_NULLS OFF, the WHERE clause will return NULL rows using this syntax. WHERE column_name <> NULL will return all non-null values.

AGGREGATION

Whenever an aggregation is used, such as SUM(), AVG(), MIN(), MAX(), and so on, NULL values will be ignored. COUNT(*) will return a count of all rows, even if one column contains NULL values, but COUNT(column_name) will omit all NULL values from the count result. GROUP BY will group NULL values, in addition to all other values.

Common Mistakes

DUMMY DATA

What happens when you create a NOT NULL column and a scenario arises where no value is entered? Consider this new column in our Accounts table, which will contain the name of the account rep:

```
ALTER TABLE Accounts ADD AccountRep VARCHAR
```

What if an account is created, but not assigned a rep yet? A dangerous mistake is to create a dummy value and populate the column with zero, blank, negative one, a default rep, or another unexpected value that easily identifies this scenario. It sounds reasonable, but is ultimately a technical burden, as you are forced to remember what the dummy value is and what it means, which can be difficult if this convention is used in multiple columns. In addition, what if a user enters your dummy value inadvertently in the web form? The result is a set of dummy data and user-entered data that cannot be effectively distinguished from each other.

Common examples where a column should probably be NULL instead of a dummy value include:

1. Social Security or National ID numbers for a non-resident. 000-00-0000 has no intrinsic meaning and doesn't provide additional information as to why that column doesn't have a valid ID number. 123-45-6789 is just as outlandish. For scenarios like this, there must be a way to indicate that the ID is inapplicable, either with NULL or some alternative flag or ID that provides meaning to the given situation.
2. End dates for ongoing tasks. 1/1/1900 is a terrible default and ultimately leads to date math errors and absurd results. 1/1/2020 and other dates in the far future are even worse, as we will one day hit those dates and be in big trouble when our dummy data has actual meaning and is comingled with dates entered by real people. If a date column is NOT NULL, it is imperative that all values entered have real meaning.
3. Text fields that are displayed directly in the application. To simplify populating all those fields and having to ISNULL() another column, we use empty strings, blanks, dashes, or other symbols as placeholders. It is a convenience for

developers, but allows for potentially lazy programming and unusual performance problems, depending on the size of the columns and the defaults that are populated for them. A database is not the optimal place to format text for display in an application. There is a slippery slope where serving data slides into serving formatted text, HTML, and other markup that probably should be left to the web tier to handle.

NULL RESULTS

It is important to account for the possibility of NULL values before you begin manipulating data. The moment you check a WHERE clause against a potentially NULL variable, you risk getting inaccurate results. The same goes for string concatenation, arithmetic, building dynamic SQL, or date math. If a column can be NULL, it's important to take the NULL possibility into account---here are some suggestions:

1. Use INNER JOIN, IF EXISTS, or WHERE clauses to filter our NULL data if it isn't needed.
2. Use ISNULL or COALESCE (if needed) to clean up the data you're returning so that we don't end up bombing future results with NULL values.
3. If lots of columns are being concatenated for frequent use in a display string, consider creating a view to simplify returning your data. This removes the need to track NULL checks in stored procedures or in code. This can be a handy tool for developers, but shouldn't be overused as views are additional objects that need to be maintained alongside the rest of our schema.

CONDITIONALS

When comparing multiple conditional statements, you can sidestep NULL problems inadvertently and not realize until it's too late that you've left behind a ticking time bomb. Consider the following SQL:

```
SELECT
       *
FROM dbo.Accounts
WHERE Address_2 = 'Apt. 5'
OR AccountName = 'Ed''s Cool Company'
```

This SQL will return any rows with the Address_2 account name specified above. But, what happens if Address_2 is NULL? This SQL will still work, but consider the following 2 queries:

```
SELECT
       *
FROM dbo.Accounts
WHERE Address_2 = 'Apt. 5'

SELECT
       *
FROM dbo.Accounts
WHERE Address_2 <> 'Apt. 5'
```

If Address_2 has a value, it will be returned by one of these queries. If Address_2 is NULL, then it will not be returned by either. If this was your intention, then you're good to go---but---if you intended for NULL values to be included in the second query, then you'd need to add a second WHERE clause to catch the NULL cases.

```
SELECT
       *
FROM dbo.Accounts
WHERE Address_2 <> 'Apt. 5' OR Accounts.Add
```

Queries that include NOT IN within the WHERE clause will exhibit the exact same behavior when dealing with a column that contains NULL values.

COUNTS

Unlike IDs, names, or other common metadata, a column that tracks a count will likely never be NULL. A count represents an absolute number of something, and in order to make sense needs to be populated with some value.

For example, let's say we have a column called PendingOrderCount that tracks the number of orders a particular account has pending. If an account has seven orders pending, then for them, PendingOrderCount = 7. If an account has no pending orders, then PendingOrderCount = 0. What if a customer has never had an order before? Generally, we'd want to use PendingOrderCount = 0 again since they have zero orders. If this column is populated using a join to some orders table, though, it's possible to inadvertently end up with a NULL for the order count, which will end up being both unexpected and undesirable.

While we could interpret NULL with special meaning in scenarios like this---that is, NULL = no orders AND never has had an order before, that would likely be confusing and hard to document. We want our developers and DBAs to actually understand how to use this data, and the necessity to document and train constantly so that quirks like this make sense should be the hint that this is probably not the best way to store a count.

As a result, counts generally should be NOT NULL columns where "zero count" and "never had any" are both entered as zero. If there is a need to report on the difference between those cases in the database, then there is almost certainly a way to do so that doesn't require the use of additional columns. In the above example, we could run an IF EXISTS against whatever table stores the order history to quickly find out with certainty if an account has ever placed any orders.

Performance

A common question that is often misquoted or answered incorrectly is, "Does a column with a NULL perform the same way as a column with an empty string?" Similarly, we can ask if a nullable column performs differently, in general, than a non-nullable column.

For non-string columns, such as INT, DATETIME, or DECIMAL, a non-NULL value has a built-in size to it. An INT column with zero instead of NULL takes 4 bytes to store, rather than 0 bytes. This much is obvious, and row sizes will increase as a result of these changes, but those are expected changes. The query optimizer knows the column is NOT NULL and how large it will be for all rows in a fixed-length column.

For a variable-length column, such as a VARCHAR column, it gets a bit tricky. NULL and empty string both take up the same space (zero bytes), but are they treated identically by the query optimizer? In general, yes---for 99.99% of queries you write, estimated row size, and therefore CPU, memory usage, and disk IO will be the same if you swap a NOT NULL '' column with a NULL column of the same data type.

It's possible to mess with the optimizer by combining VARCHAR(MAX) and complex queries with a variety of hash joins and ORDER BY clauses. The result can be hash joins and worktables with overly inflated sizes, resulting directly from unusually large estimated row sizes when a NOT NULL empty string column is used vs. a NULLable column with NULL values entered instead. This isn't the norm, though, and I mention it only as a venue for troubleshooting in the event that a very complex query involving a table such as this is resulting in performance problems and heavy tempDB pressure.

Beyond this edge case, performance shouldn't be a concern in deciding NULL vs. NOT NULL in a variable length string. It still matters for other data types, where an empty string isn't available as zero-length filler.

Conclusion

A great deal can be said about what seems like such a simple topic. NULL is special in the world of database design, and understanding how it works and is used in the context of your data will go a long way towards ensuring that you make the best decisions possible. Those smart decisions early on will prove valuable years later when all of the TSQL that has been written based on them is intuitive, scalable, and easy to update.