NULLs Make Things Easier?

BRUCE MOMJIAN

Nulls are a very useful but also very error-prone relational database feature. This talk is designed to help applications developers better manage their use of nulls.

https://momjian.us/presentations

Last updated: October 2023

Nulls in English

Null means “nothing”.

Nulls In Computer Languages

C-based languages use a NULL pointer to indicate a pointer that does not point to a value. Languages that don't use pointers often use an "undefined" value for a similar purpose.

Nulls in Data

What do you place in a field that has no value?
For strings, a zero-length string is reasonable.
What about numerics? -1, -99, 0?
What about dates? 1900-01-01?

Why Use NULLs

The three meanings of NULL:

- Unknown values
- Inapplicable values
- Empty placeholders

The NULL Spouse Example

If *employee.spouse* is NULL, does it mean?

- The spouse's name is unknown.
- The employee is not married and therefore has no spouse.
- The *employee.spouse* column was an unjoined column from an outer join.

NULLs Can Cause Their Own Problems

Don't use NULLs in inappropriate situations.

https://www.flickrl.com/photos/randar/

Warning!

In their book A Guide to Sybase and SQL Server, David McGoveran and C. J. Date said:

> *It is this writer's opinion than NULLs, at least as currently defined and implemented in SQL, are far more trouble than they are worth and should be avoided; they display very strange and inconsistent behavior and can be a rich source of error and confusion. (Please note that these comments and criticisms apply to any system that supports SQL-style NULLs, not just to SQL Server specifically.)"*
>
> ...
>
> *In the rest of this book, I will be urging you not to use them, which may seem contradictory, but it is not. Think of a NULL as a drug; use it properly and it works for you, but abuse it and it can ruin everything. Your best policy is to avoid NULLs when you can and use them properly when you have to.*

Joe Celko, *SQL for Smarties: Advanced SQL Programming*

Keep Your Eye on the Red (Text)

https://www.flickr.com/photos/alltheaces/

Explicit NULLs

```
test=> SELECT NULL;
 ?column?
----------

test=> \pset null (null)

test=> SELECT NULL;
 ?column?
----------
 (null)
```

All queries in this presentation can be downloaded from `https://momjian.us/main/writings/pgsql/nulls.sql`.

Explicitly NULL Assignment

```
CREATE TABLE nulltest (x INTEGER, y INTEGER);

INSERT INTO nulltest VALUES (1, NULL);

SELECT * FROM nulltest;
 x |   y
---+--------
 1 | (null)
```

Implicit NULL Assignment

```
INSERT INTO nulltest (x) VALUES (2);

SELECT * FROM nulltest;
 x |   y
---+--------
 1 | (null)
 2 | (null)
```

NULL Storage Can Be Prevented

```
CREATE TABLE nulltest2 (x INTEGER NOT NULL, y INTEGER NOT NULL);

INSERT INTO nulltest2 VALUES (3, NULL);
ERROR:  null value in column "y" violates not-null constraint
DETAIL:  Failing row contains (3, null).

INSERT INTO nulltest2 (x) VALUES (4);
ERROR:  null value in column "y" violates not-null constraint
DETAIL:  Failing row contains (4, null).
```

The Non-Value of NULLs

```
SELECT NULL + 1;
 ?column?
----------
 (null)

SELECT NULL || 'a';
 ?column?
----------
 (null)

SELECT 'b' || NULL;
 ?column?
----------
 (null)
```

NULL Is Unknown?

```
CREATE TABLE inctest (x INTEGER);

INSERT INTO inctest VALUES (30), (40), (NULL);

SELECT x + 1 FROM inctest;
 ?column?
----------
       31
       41
   (null)
```

The Three-Valued Logic of NULLs

```
SELECT NULL = 1;
 ?column?
----------
 (null)

SELECT NULL = '';
 ?column?
----------
 (null)

SELECT NULL = NULL;
 ?column?
----------
 (null)

SELECT NULL < NULL + 1;
 ?column?
----------
 (null)
```

NULL represents unknown, not applicable, or unassigned. It has no data type, so comparing it to fixed values always returns NULL.

NULL Query Comparisons

```
SELECT 1
WHERE true;
 ?column?
----------
        1

SELECT 1
WHERE false;
 ?column?
----------

SELECT 1
WHERE NULL;
 ?column?
----------
```

WHERE only returns rows whose result is *true*, not *false* or NULL.

NULL Is Not False

```
SELECT true AND NULL;
 ?column?
----------
 (null)

SELECT NOT NULL;
 ?column?
----------
 (null)
```

NULL Operator Comparisons

```
SELECT * FROM inctest;
   x
--------
     30
     40
 (null)

SELECT * FROM inctest WHERE x >= 0;
 x
----
 30
 40

SELECT * FROM inctest WHERE x < 0;
 x
----
```

NULL And Not Equals

```
SELECT * FROM inctest WHERE x < 0 OR x >= 0;
 x
----
 30
 40

SELECT * FROM inctest WHERE x <> 0;
 x
----
 30
 40
```

Nulls And Not In

```
SELECT 1 <> 2 AND 1 <> 3;
 ?column?
----------
 t

SELECT 1 <> 2 AND 1 <> 3 AND 1 <> NULL;
 ?column?
----------
 (null)
```

Subqueries With NULL

```
SELECT 'a' IN (SELECT NULL::text);
 ?column?
----------
 (null)

SELECT 'a' NOT IN (SELECT NULL::text);
 ?column?
----------
 (null)
```

Multi-Row Subqueries

```
SELECT 'a' IN (VALUES ('a'), (NULL));
 ?column?
----------
 t

SELECT 'a' NOT IN (VALUES ('a'), (NULL));
 ?column?
----------
 f

SELECT 'a' IN (VALUES ('b'), (NULL));
 ?column?
----------
 (null)

SELECT 'a' NOT IN (VALUES ('b'), (NULL));
 ?column?
----------
 (null)
```

IN Queries Expanded

```
SELECT 'a' = 'b' OR 'a' = NULL;
 ?column?
----------
 (null)

SELECT 'a' <> 'b' AND 'a' <> NULL;
 ?column?
----------
 (null)
```

NOT IN subqueries returning NULLs are often problematic.

EXISTS Queries

```
SELECT EXISTS (SELECT 1);
 exists
--------
 t
SELECT EXISTS (SELECT NULL);
 exists
--------
 t
SELECT NOT EXISTS (SELECT NULL);
 ?column?
----------
 f
```

EXISTS handles NULL values differently than IN because EXISTS/NOT EXISTS only considers whether rows are returned from subqueries; it is not matching values.

Explicit NULL Comparison

```
SELECT NULL = NULL;
 ?column?
----------
 (null)

SELECT NULL IS NULL;
 ?column?
----------
 t

SELECT NULL IS NOT NULL;
 ?column?
----------
 f
```

Explicit NULL Comparison

```
SELECT * FROM inctest;
   x
--------
     30
     40
 (null)

SELECT * FROM inctest WHERE x IS NULL;
   x
--------
 (null)

SELECT * FROM inctest WHERE x IS NOT NULL;
 x
----
 30
 40
```

Comparing NULLs With True/False Logic

```
SELECT 1 IS NOT DISTINCT FROM 1;
 ?column?
----------
 t

SELECT NULL = 1;
 ?column?
----------
 (null)

SELECT NULL IS NOT DISTINCT FROM 1;
 ?column?
----------
 f

SELECT NULL IS NOT DISTINCT FROM NULL;
 ?column?
----------
 t
```

Explicit Equality Comparisons With NULL

```
SELECT * FROM inctest WHERE x IS DISTINCT FROM 30;
   x
--------
 40
 (null)

SELECT * FROM inctest WHERE x IS NOT DISTINCT FROM 30;
 x
----
 30
```

Comparing NULLs to Other NULLs

```
CREATE TABLE disttest (x INTEGER, y INTEGER);

INSERT INTO disttest VALUES (1, 1), (2, 3), (NULL, NULL);

SELECT * FROM disttest where x IS NOT DISTINCT FROM y;
   x    |   y
--------+--------
      1 |      1
 (null) | (null)
```

This is particularly useful for joins.

Ordering of NULLs

```
SELECT * FROM (VALUES (NULL), (2), (1), (NULL)) AS v(x) ORDER BY 1;
   x
--------
      1
      2
 (null)
 (null)
```

NULLs are treated as equal for ordering purposes.

Ordering NULLs First

```
SELECT * FROM (VALUES (NULL), (2), (1), (NULL)) AS v(x)
ORDER BY 1 NULLS FIRST;
   x
--------
 (null)
 (null)
      1
      2
```

Unique Indexes Treat NULLs as Unequal by Default

```
CREATE TABLE uniqtest (x INTEGER);

CREATE UNIQUE INDEX i_uniqtest ON uniqtest (x);

INSERT INTO uniqtest VALUES (1), (NULL), (NULL);

SELECT * FROM uniqtest;
   x
--------
      1
 (null)
 (null)
```

This can be changed in Postgres 15 and later with NULLS DISTINCT.

CHECK Constraints Allow NULLs

```
CREATE TABLE checktest (x INTEGER CHECK (x > 0));

INSERT INTO checktest VALUES (1), (NULL);

SELECT * FROM CHECKTEST;
   x
--------
      1
 (null)

ALTER TABLE checktest ALTER COLUMN x SET NOT NULL;
ERROR:  column "x" of relation "checktest" contains null values
```

NULLs and Aggregates

```
CREATE TABLE aggtest (x INTEGER);

INSERT INTO aggtest VALUES (7), (8), (NULL);

SELECT COUNT(*), COUNT(x), SUM(x), MIN(x), MAX(x), AVG(x) FROM aggtest;
 count | count | sum | min | max |         avg
-------+-------+-----+-----+-----+--------------------
     3 |     2 |  15 |   7 |   8 | 7.5000000000000000

DELETE FROM aggtest;

SELECT COUNT(*), COUNT(x), SUM(x), MIN(x), MAX(x), AVG(x) FROM aggtest;
 count | count |  sum   |  min   |  max   |  avg
-------+-------+--------+--------+--------+--------
     0 |     0 | (null) | (null) | (null) | (null)
```

The sum of zero rows is NULL.

NULLs and GROUP BY

```
DELETE FROM aggtest;

INSERT INTO aggtest VALUES (7), (8), (NULL), (NULL);

SELECT x, COUNT(*), COUNT(x), SUM(x), MIN(x), MAX(x), AVG(x)
FROM aggtest
GROUP BY x
ORDER BY x;
   x    | count | count |  sum   |  min   |  max   |        avg
--------+-------+-------+--------+--------+--------+--------------------
      7 |     1 |     1 |      7 |      7 |      7 | 7.0000000000000000
      8 |     1 |     1 |      8 |      8 |      8 | 8.0000000000000000
 (null) |     2 |     0 | (null) | (null) | (null) |             (null)
```

Mapping NULLs to Non-NULLs

```
SELECT COALESCE(NULL, 0);
 coalesce
----------
        0

SELECT COALESCE(NULL, 'I am null.');
  coalesce
------------
 I am null.
```

Mapping NULLs to Non-NULLs

```
CREATE TABLE nullmaptest (x TEXT);

INSERT INTO nullmaptest VALUES ('f'), ('g'), (NULL);

SELECT x, COALESCE(x, 'n/a') FROM nullmaptest;
   x    | coalesce
--------+----------
 f      | f
 g      | g
 (null) | n/a

SELECT 'a' || COALESCE(NULL, '') || 'b';
 ?column?
----------
 ab

SELECT SUM(x), COALESCE(SUM(x), 0) FROM aggtest;
  sum   | coalesce
--------+----------
 (null) |        0
```

Mapping NULLs to Non-NULLs

```
DELETE FROM nullmaptest;

INSERT INTO nullmaptest VALUES ('f'), ('g'), ('n/a');

SELECT x, NULLIF(x, 'n/a') FROM nullmaptest;
  x  | nullif
-----+--------
 f   | f
 g   | g
 n/a | (null)

SELECT NULLIF('n/a', COALESCE(NULL, 'n/a'));
 nullif
--------
 (null)
```

NULLs In Arrays

```
SELECT NULL::INTEGER[] IS NULL;
 ?column?
----------
 t

SELECT '{}'::INTEGER[] IS NULL;
 ?column?
----------
 f

SELECT '{NULL}'::INTEGER[] IS NULL;
 ?column?
----------
 f
```

Row Expressions With NULLs

```
SELECT ROW() IS NULL, ROW() IS NOT NULL;
 ?column? | ?column?
----------+----------
 t        | t

SELECT ROW(NULL) IS NULL;
 ?column?
----------
 t

SELECT ROW(NULL, NULL) IS NULL;
 ?column?
----------
 t
```

Row Expressions With NULLs

```
SELECT ROW(NULL, 1) IS NULL;
 ?column?
----------
 f

SELECT ROW(NULL, 1) IS NOT NULL;
 ?column?
----------
 f

SELECT ROW(1, 2) IS NOT NULL;
 ?column?
----------
 t
```

Queries Returning NULLs in the Target List

```
CREATE TABLE emptytest (x INTEGER);

SELECT * from emptytest;
 x
----

SELECT (SELECT * from emptytest);
   x
--------
 (null)

SELECT (SELECT * from emptytest) IS NULL;
 ?column?
----------
 t
```

A SELECT with no FROM clause is assumed to return one row.

I Think I Get It!

"Oh, that makes sense" — When you see individual behaviors of null, they look systematic, and your brain quickly sees a pattern and extrapolates what might happen in other situations. Often, that extrapolation is wrong, because null semantics are a mix of behaviors. I think the best way to think about null is as a Frankenstein monster of several philosophies and systems stitched together by a series of special cases.

Jeff Davis

Tips for Taming NULLs

- Define columns with NOT NULL constraints where appropriate
  - preventing inappropriate NULL storage avoids future problems
  - consider preventing zero-length strings too; domains can be used to combine constraints
- Consider three-valued logic in expressions
  - Avoid the return of NULL values from NOT IN subqueries
    - perhaps add *col* IS NOT NULL
  - Use IS NOT DISTINCT FROM for equality comparisons with possible NULL values
    - *col1* IS NOT DISTINCT FROM *col2* is the same as *col1* = *col2* OR (*col1* IS NULL AND *col2* IS NULL)
  - IS DISTINCT FROM is used for not-equal comparisons
- Use COALESCE() to map NULL values to non-NULL values and NULLIF() for the reverse

Conclusion

The presentation blog posts are at `https://momjian.us/main/blogs/pgblog/2013.html#January_23_2013`.

https://momjian.us/presentations

https://www.flickr.com/photos/micspecial/inglefttosayccd/