NULLs and Three-Valued Logic: The Source of Countless Bugs

NULL Is Not a Value

NULL means "unknown" or "not applicable."

It's not:

- An empty string (`''`)
- Zero (`0`)
- False (`false`)

It's the absence of a value.

This distinction breaks intuition for developers coming from most programming languages.

Three-Valued Logic

In most languages, boolean expressions are **binary**: TRUE or FALSE.

In SQL, they're **ternary**: TRUE, FALSE, or **UNKNOWN**.

```
SELECT * FROM users WHERE email = NULL;  -- Returns 0 rows!
```

Why? `email = NULL` evaluates to UNKNOWN, not TRUE.

UNKNOWN rows are **filtered out** in WHERE clauses.

Truth Tables

AND

A	B	A AND B
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
TRUE	UNKNOWN	UNKNOWN
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE
FALSE	UNKNOWN	FALSE
UNKNOWN	TRUE	UNKNOWN
UNKNOWN	FALSE	FALSE
UNKNOWN	UNKNOWN	UNKNOWN

Key insight: `FALSE AND UNKNOWN = FALSE` (short-circuit).

OR

A	B	A OR B
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	UNKNOWN	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE
FALSE	UNKNOWN	UNKNOWN
UNKNOWN	TRUE	TRUE
UNKNOWN	FALSE	UNKNOWN
UNKNOWN	UNKNOWN	UNKNOWN

Key insight: `TRUE OR UNKNOWN = TRUE` (short-circuit).

NOT

A	NOT A
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

Key insight: `NOT UNKNOWN = UNKNOWN` (not FALSE!).

Comparing to NULL: Always UNKNOWN

Equality

```sql
SELECT * FROM users WHERE email = NULL;  -- Wrong!
```

Result: 0 rows, always.

Why? `email = NULL` is UNKNOWN (even if email IS NULL).

Correct:

```sql
SELECT * FROM users WHERE email IS NULL;
```

`IS NULL` is a special operator that returns TRUE/FALSE (not UNKNOWN).

Inequality

```sql
SELECT * FROM users WHERE email != NULL;  -- Wrong!
```

Result: 0 rows.

Why? `email != NULL` is UNKNOWN.

Correct:

```sql
SELECT * FROM users WHERE email IS NOT NULL;
```

The Trap: NULL in Subqueries

```sql
SELECT * FROM orders WHERE user_id NOT IN (SELECT id FROM users WHERE active =
false);
```

If any user.id is NULL, this returns 0 rows!

Why?

- `user_id NOT IN (1, 2, NULL)` expands to:
- `user_id != 1 AND user_id != 2 AND user_id != NULL`
- `user_id != NULL` is UNKNOWN
- `TRUE AND TRUE AND UNKNOWN = UNKNOWN`
- UNKNOWN rows are filtered out

Fix: Exclude NULLs explicitly:

```sql
SELECT * FROM orders WHERE user_id NOT IN (SELECT id FROM users WHERE active = false
AND id IS NOT NULL);
```

Or use `NOT EXISTS` (safer):

```sql
SELECT * FROM orders o
WHERE NOT EXISTS (
  SELECT 1 FROM users u WHERE u.id = o.user_id AND u.active = false
);
```

NULL in Aggregates

COUNT: The Special Case

```sql
SELECT
  COUNT(*),      -- Counts rows
  COUNT(email),  -- Counts non-NULL emails
  COUNT(*)
FROM users;
```

If `email` is nullable:

- `COUNT(*)` = total rows (including NULLs)
- `COUNT(email)` = rows with non-NULL email

Different values!

SUM, AVG: Ignore NULLs

```
SELECT AVG(rating) FROM reviews;
```

Data: ``` rating

5 4 NULL 3

```
**Result:** `AVG(rating) = 4` (not 3, not 2.4).

**Why?** NULLs are excluded from both numerator and denominator.

- Sum: 5 + 4 + 3 = 12
- Count: 3 (not 4)
- Avg: 12 / 3 = 4

### MIN, MAX: Ignore NULLs

```sql
SELECT MIN(price), MAX(price) FROM products;
```

If all prices are NULL: Returns NULL.

NULL Propagation in Expressions

```
SELECT 1 + NULL;       -- Returns NULL
SELECT 'Hello' || NULL; -- Returns NULL
SELECT NULL * 100;     -- Returns NULL
```

Rule: Any arithmetic or string operation involving NULL returns NULL (except COALESCE).

COALESCE: The NULL Handler

COALESCE returns the first non-NULL value:

```
SELECT COALESCE(email, phone, 'No contact') AS contact FROM users;
```

If `email` is NULL but `phone` is not: Returns `phone`.

If both are NULL: Returns `'No contact'`.

Common Uses

Default Values
```

```
SELECT name, COALESCE(discount, 0) AS discount FROM products;
```

If discount is NULL: Returns 0.

Safe Aggregation

```
SELECT user_id, COALESCE(SUM(amount), 0) AS total FROM transactions GROUP BY
user_id;
```

If a user has no transactions (or all amounts are NULL): Returns 0 instead of NULL.

COALESCE vs IFNULL / NVL

SQL Standard: `COALESCE(a, b, c)`

MySQL: `IFNULL(a, b)` (only 2 arguments)

Oracle: `NVL(a, b)` (only 2 arguments)

Use COALESCE for portability.

NULLIF: The Reverse of COALESCE

NULLIF returns NULL if two values are equal:

```
SELECT NULLIF(value, 0) FROM table;
```

If value = 0: Returns NULL.

If value != 0: Returns value.

Use case: Avoiding division by zero:

```
SELECT total / NULLIF(count, 0) AS average FROM stats;
```

If `count = 0`, division returns NULL (not error).

NULL in JOINs

LEFT JOIN: NULLs Indicate Missing Matches

```
SELECT u.name, o.id AS order_id
FROM users u
LEFT JOIN orders o ON u.id = o.user_id;
```

If a user has no orders:

```
name    | order_id
--------|----------
```

```
Alice   | 101
Bob     | NULL      ← Bob has no orders
```

Filtering NULLs in WHERE turns LEFT JOIN into INNER JOIN:

```
SELECT u.name, o.id
FROM users u
LEFT JOIN orders o ON u.id = o.user_id
WHERE o.id IS NOT NULL;  -- Excludes users without orders
```

This is effectively an INNER JOIN. Use INNER JOIN instead for clarity.

NULL as a Join Key: Doesn't Match

```
SELECT * FROM users u
JOIN orders o ON u.optional_field = o.optional_field;
```

If both fields are NULL: They **don't match**.

Why? `NULL = NULL` is UNKNOWN, not TRUE.

Workaround (if you want NULL to match NULL):

```
ON (u.optional_field = o.optional_field OR (u.optional_field IS NULL AND
o.optional_field IS NULL))
```

Or use `IS NOT DISTINCT FROM` (Postgres):

```
ON u.optional_field IS NOT DISTINCT FROM o.optional_field
```

NULL in ORDER BY

Default Behavior

Postgres: NULLs sort **last** (ascending) or **first** (descending).

MySQL: NULLs sort **first** (ascending) or **last** (descending).

Example:

```
SELECT name, rating FROM products ORDER BY rating;
```

Postgres:

```
name    | rating
--------|--------
A       | 1
B       | 2
C       | NULL
```

MySQL:

```
name    | rating
--------|--------
C       | NULL
A       | 1
B       | 2
```

Explicit NULL Handling

Postgres:

```sql
ORDER BY rating NULLS FIRST;
ORDER BY rating NULLS LAST;
```

MySQL: Use conditional sorting:

```sql
ORDER BY rating IS NULL, rating;  -- NULLs last
```

NULL in DISTINCT

NULL values are considered duplicates:

```sql
SELECT DISTINCT email FROM users;
```

If 3 users have NULL email: Only one NULL appears in the result.

This is inconsistent with `NULL = NULL` (which is UNKNOWN), but pragmatic.

NULL in UNIQUE Constraints

Postgres: Multiple NULLs Allowed

```sql
CREATE TABLE products (
  id INT PRIMARY KEY,
  sku TEXT UNIQUE
);

INSERT INTO products (id, sku) VALUES (1, NULL);
INSERT INTO products (id, sku) VALUES (2, NULL);  -- Allowed!
```

Why? NULL != NULL, so no uniqueness violation.

SQL Standard: Same behavior (Postgres follows it)

MySQL: Depends on storage engine

- **InnoDB:** Multiple NULLs allowed (like Postgres).
- **MyISAM:** Multiple NULLs allowed.

Workaround: Use COALESCE with a sentinel value

```
CREATE UNIQUE INDEX idx_products_sku ON products (COALESCE(sku, ''));
```

Now only one NULL (treated as `''`) is allowed.

NULL in CHECK Constraints

CHECK constraints allow NULL:

```
CREATE TABLE users (
  age INT CHECK (age >= 18)
);

INSERT INTO users (age) VALUES (NULL);  -- Allowed!
```

Why? `NULL >= 18` is UNKNOWN, and CHECK constraints allow UNKNOWN.

If you want to enforce non-NULL:

```
age INT NOT NULL CHECK (age >= 18)
```

Interview Trap Examples

Trap 1: Counting NULLs

Question: How many users have no email?

Wrong:

```
SELECT COUNT(email) FROM users WHERE email = NULL;
```

Result: 0 (always).

Right:

```
SELECT COUNT(*) FROM users WHERE email IS NULL;
```

Or:

```
SELECT COUNT(*) - COUNT(email) FROM users;
```

Trap 2: NOT IN with NULLs

Question: Find orders not placed by inactive users.

Wrong:

```
SELECT * FROM orders WHERE user_id NOT IN (SELECT id FROM users WHERE active =
false);
```

If any inactive user has `id = NULL`, this returns 0 rows!

Right:

```
SELECT * FROM orders o
WHERE NOT EXISTS (
  SELECT 1 FROM users u WHERE u.id = o.user_id AND u.active = false
);
```

Trap 3: NULL in Arithmetic

Question: Calculate total revenue.

Naive:

```
SELECT SUM(quantity * price) FROM order_items;
```

Problem: If any `quantity` or `price` is NULL, that row contributes NULL (ignored by SUM, but might not be intended).

Safe:

```
SELECT SUM(COALESCE(quantity, 0) * COALESCE(price, 0)) FROM order_items;
```

Or enforce NOT NULL constraints.

Trap 4: WHERE col != value Doesn't Include NULLs

Question: Find all products not in category 'Books'.

Wrong:

```
SELECT * FROM products WHERE category != 'Books';
```

Problem: Products with `category = NULL` are excluded!

Why? `NULL != 'Books'` is UNKNOWN.

Right:

```
SELECT * FROM products WHERE category != 'Books' OR category IS NULL;
```

Or:

```
SELECT * FROM products WHERE NOT (category = 'Books');
```

(But this is confusing. Explicit IS NULL is clearer.)

Trap 5: MAX/MIN with All NULLs

Data: ``` price

NULL NULL

```sql
SELECT MAX(price) FROM products;
```

Result: NULL (not 0, not error).

Safe:

```sql
SELECT COALESCE(MAX(price), 0) FROM products;
```

Practical NULL Handling Strategies

Strategy 1: Avoid NULLs When Possible

Use NOT NULL + DEFAULT:

```sql
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  name TEXT NOT NULL,
  email TEXT NOT NULL,
  phone TEXT NOT NULL DEFAULT '',
  active BOOLEAN NOT NULL DEFAULT true
);
```

Benefits:

- No NULL checks needed
- Clearer semantics
- Fewer bugs

When NULL is appropriate:

- Optional foreign keys
- Truly unknown values (e.g., end_date for ongoing events)

Strategy 2: Use COALESCE for Defaults

```sql
SELECT name, COALESCE(discount, 0) AS discount FROM products;
```

Instead of checking IS NULL everywhere.

Strategy 3: Use NOT EXISTS Instead of NOT IN

Safer:

```
WHERE NOT EXISTS (SELECT 1 FROM other_table WHERE ...)
```

Than:

```
WHERE id NOT IN (SELECT id FROM other_table)
```

Strategy 4: Explicit NULL Handling in Application

Don't assume SQL will do what you expect:

```
const result = await db.query('SELECT email FROM users WHERE id = $1', [userId]);

if (result.rows[0].email === null) {
  // Handle NULL explicitly
  email = 'no-email@example.com';
} else {
  email = result.rows[0].email;
}
```

Strategy 5: Use Partial Indexes to Exclude NULLs

```
CREATE INDEX idx_users_phone ON users(phone) WHERE phone IS NOT NULL;
```

Benefits:

- Smaller index (doesn't index NULLs)
- Faster queries on non-NULL values

NULL vs Empty String vs Zero

Empty String (`''`)

Is a value. Not the same as NULL.

```
SELECT '' IS NULL;       -- FALSE
SELECT '' = '';          -- TRUE
SELECT LENGTH('');       -- 0
```

Use case: Explicitly empty text fields.

Zero (`0`)

Is a value. Not the same as NULL.

```
SELECT 0 IS NULL;        -- FALSE
SELECT 0 = 0;            -- TRUE
SELECT 0 + 10;           -- 10
```

Use case: Numeric zero (count, balance, etc.).

NULL

Is not a value.

```
SELECT NULL IS NULL;      -- TRUE
SELECT NULL = NULL;       -- UNKNOWN (!)
SELECT NULL + 10;         -- NULL
```

Use case: Unknown or inapplicable values.

Key Takeaways

1. **NULL means "unknown," not a value.** It's not 0, not `''`, not false.
2. **Comparing to NULL with `=` is always UNKNOWN.** Use `IS NULL` / `IS NOT NULL`.
3. **Three-valued logic:** TRUE, FALSE, UNKNOWN. WHERE filters out UNKNOWN.
4. **COUNT(*) counts all rows; COUNT(column) counts non-NULL values.**
5. **Aggregates (SUM, AVG, MIN, MAX) ignore NULLs.**
6. **NULL propagates in expressions:** `1 + NULL = NULL`.
7. **COALESCE returns the first non-NULL value.**
8. **NOT IN with NULLs is dangerous.** Use NOT EXISTS.
9. **LEFT JOIN produces NULLs for missing matches.** Don't filter them in WHERE unless you mean INNER JOIN.
10. **Avoid NULLs when possible** with NOT NULL + DEFAULT.

Next up: Constraints and schema design—how to make the database enforce correctness.