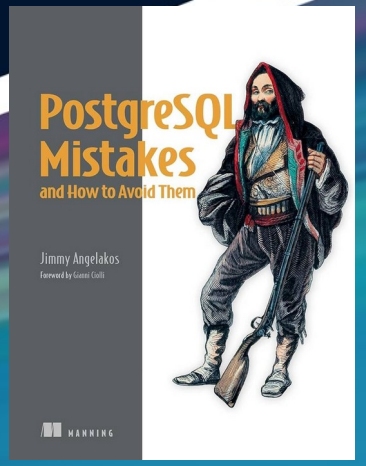
Live: Fixing Bad SQL in PostgreSQL

Jimmy Angelakos
Staff Software Engineer

2026-04-03



MANNING

pgEdge®

About Jimmy Angelakos

- Systems & Database Architect, based in Edinburgh, Scotland
- Involved in the Open Source Community for 25+ years
- PostgreSQL Significant Contributor
- Member, PostgreSQL Europe Diversity Committee
- Author, PostgreSQL Mistakes and How to Avoid Them
- pg_statviz PostgreSQL extension

The Danger of Silent Bugs

- SQL doesn't crash when it's wrong — it returns a result
- Wrong results look like right results
- Performance degrades gradually, then all at once
- Today: 10 anti-patterns from Chapter 2, live on a real dataset

The Dataset — Frogge Emporium

- ERP system: `customers`, `orders`, `invoices`, `payments`, `suppliers`
- Support system: `support.tickets` (1 million+ rows)
- ~14,000 customers, ~250,000 orders, ~250,000 invoices
- 2 suppliers — one in Michigan, one in Japan (`state = NULL`)
- Generated from `github.com/vyruss/postgresql-mistakes`

Session Format

For each anti-pattern:

- Show the **broken query** — looks reasonable
- Show **why it's wrong** — the silent failure
- **Refactor** — the correct version
- `EXPLAIN ANALYZE` to see the performance difference

The NOT IN Trap — The Scenario

- Frogge Emporium wants to run a promotion
- Need emails of customers in states **without** a supplier
- Supplier table has 2 rows:
  - Omni Consumer Products — `state = 'MI'`
  - Yoyodyne — `state = NULL` (Japan)

The NOT IN Trap — The Broken Query

```
SELECT email
FROM erp.customer_contact_details
WHERE state NOT IN (SELECT state
                    FROM erp.suppliers);
```

Result: 0 rows !

But we KNOW there are customers in Kansas...

```
SELECT email FROM erp.customer_contact_details
WHERE state = 'KS' LIMIT 1;
 -- river.smith@example.com
```

Why NOT IN Breaks with NULLs

- Subquery returns `('MI', NULL)`
- `'KS' NOT IN ('MI', NULL)` evaluates as:
  - `'KS' <> 'MI'` → TRUE
  - `'KS' <> NULL` → UNKNOWN
  - `TRUE AND UNKNOWN` → UNKNOWN
- `WHERE UNKNOWN` is treated as FALSE — row excluded
- One NULL poisons the **entire result set** to zero rows
- Python: `1 not in [2, None]` is `True.` SQL is NOT Python !

The Fix — NOT EXISTS

```
SELECT ccd.email
FROM erp.customer_contact_details ccd
WHERE NOT EXISTS (SELECT FROM erp.suppliers s
                  WHERE ccd.state = s.state)
AND ccd.state IS NOT NULL ;
```

Alternative: LEFT JOIN / IS NULL

```
SELECT ccd.email
FROM erp.customer_contact_details ccd
LEFT JOIN erp.suppliers s USING (state)
WHERE s.state IS NULL
AND ccd.state IS NOT NULL;
```

NOT IN — Takeaway

- `NOT IN (subquery)` — **Not NULL-safe** — silently returns 0 rows
- `NOT EXISTS` — **NULL-safe** — Anti Join plan
- `LEFT JOIN ... IS NULL` — **NULL-safe** — Anti Join plan

Rule: Never use NOT IN with a subquery that can contain NULLs.

BETWEEN on Timestamps — The Scenario

- Frogge Emporium sums yesterday's payments every morning
- They use `BETWEEN` to define the date range

BETWEEN — The Broken Query

```
WITH t(today) AS (SELECT CURRENT_DATE::timestamptz)
SELECT sum (amount)
FROM erp.payments , t
WHERE tstamp BETWEEN t.today - INTERVAL '1 day'
                 AND t.today;
```

Yesterday's LAST payment and today's FIRST payment are the **same timestamp** — double-counted !

BETWEEN — The Problem

- `BETWEEN` is inclusive on **both** ends
- A payment at exactly midnight gets counted **twice**
- When dealing with money, double-counting is serious

BETWEEN — The Fix

```
-- Half-open interval: exclusive upper bound
 WITH t(today) AS (SELECT CURRENT_DATE::timestamptz)
 SELECT sum(amount), count(amount)
 FROM erp.payments, t
 WHERE tstamp >= t.today - INTERVAL '1 day'
   AND tstamp < t.today ;
```

Rule: `WHERE ts >= start AND ts < end`

- No boundary ambiguity
- Adjacent ranges never overlap or gap
- Works for any precision

Dividing INTEGERs

```
WITH
i AS (SELECT count (*) c FROM erp.orders
      WHERE item IS NOT NULL ),
s AS (SELECT count (*) c FROM erp.orders
      WHERE service IS NOT NULL )
SELECT
    i.c / ( i.c + s.c ) * 100 AS "Item %" ,
    s.c / ( i.c + s.c ) * 100 AS "Service %"
FROM i , s ;
```

Result: Item % = 0, Service % = 0

Integer / Integer truncates toward zero !

The Fix — Cast to Float

```
WITH
i AS (SELECT count(*)::float c FROM erp.orders
      WHERE item IS NOT NULL),
s AS (SELECT count(*)::float c FROM erp.orders
      WHERE service IS NOT NULL)
SELECT
    round ((i.c / (i.c + s.c) * 100)::numeric, 1) AS "Item %" ,
    round ((s.c / (i.c + s.c) * 100)::numeric, 1) AS "Service %"
FROM i, s ;
```

Result: **Item % = 99.5, Service % = 0.5**

Guard against division by zero with `NULLIF(denominator, 0)`

COUNTing NULL Values

```
SELECT count (item) FROM erp.orders
WHERE placed_at > date_trunc ('year', CURRENT_DATE);
 -- Returns 198,920 (not 200,000!)
```

~1,080 orders are services (`item = NULL`). `COUNT(column)` ignores NULLs.

```
SELECT count(*) FROM erp.orders
WHERE placed_at > date_trunc ('year', CURRENT_DATE);
 -- Returns 200,000 (correct total)
```

The Rules for counting NULLs

- **Always cast before dividing** when you need a fraction
- **Guard against division by zero** with `NULLIF(denom, 0)`
- **Choose your COUNT intentionally:**
  - `COUNT(*)` = total rows
  - `COUNT(column)` = non-NULL values only
  - You can exploit this: `COUNT(nullable)` deliberately skips NULLs

Upserting NULLs in a Composite Unique Key

Frogge Emporium's inventory system:

- Product can exist in multiple warehouses
- Within a warehouse, can be in a specific area (freezer)
- Or in the common stock area (`area = NULL`)
- Unique key: `(product_id, warehouse_id, area)`

The Broken Upsert

```
-- Insert to common area (area = NULL), quantity 5
INSERT INTO erp.inventory (product_id, warehouse_id, area, quantity)
VALUES (99999, 1, null, 5)
ON CONFLICT (product_id, warehouse_id, area ) DO UPDATE
SET quantity = EXCLUDED.quantity ;


-- Update quantity to 7:
INSERT INTO erp.inventory (product_id, warehouse_id, area, quantity)
VALUES (99999, 1, null, 7)
ON CONFLICT (product_id, warehouse_id, area)
DO UPDATE SET quantity = EXCLUDED.quantity ;
```

Result: 3 rows ! Two with area=NULL.

The upsert INSERTED a duplicate instead of updating.

Why This Happens

- In SQL: `NULL = NULL` is **UNKNOWN** , not TRUE
- Unique index treats each NULL as **distinct**
- `(99999, 1, NULL)` and `(99999, 1, NULL)` are "different"
- `ON CONFLICT` never fires — Postgres sees no conflict
- Before PostgreSQL 14, behavior was even **unpredictable**

The Fix — NULLS NOT DISTINCT (PG 15+)

```
CREATE UNIQUE INDEX ON erp.inventory
(product_id, warehouse_id, area )
NULLS NOT DISTINCT ;
```

- Now the upsert correctly updates the NULL-area row
- Older versions: expression index with `COALESCE(area, 'common_area')`
- Best option: redesign column as NOT NULL with explicit default

Querying Indexed Columns with Expressions

- `erp.payments` has an index on `tstamp`
- Exact timestamp lookup: **Index Scan — 0.022 ms**
- Developer uses `date_trunc()` for second-level accuracy...

The Broken Query

```
EXPLAIN (ANALYZE)
SELECT * FROM erp.payments
WHERE date_trunc('s' , tstamp) = '2023-10-18 03:40:34';
```

- Plan: **Seq Scan**
- Execution Time: **35 ms**
- Rows Removed by Filter: **249,999**
- That's **1,600x SLOWER** than the index scan !

Why Wrapping Kills the Index

- Index stores raw `tstamp` values (timestamptz)
- `date_trunc('s', tstamp)` is a **function call**
- Postgres must evaluate it for **every single row**
- The planner cannot look inside the function
- You're paying for the index without getting any benefit

Fix 1 — Keep the Column Bare

```
EXPLAIN (ANALYZE)
SELECT * FROM erp.payments
WHERE tstamp >= '2023-10-18 03:40:34'::timestamptz
AND tstamp < '2023-10-18 03:40:34'::timestamptz
    + INTERVAL '1 s' ;
```

Plan: **Index Scan** using `payments_tstamp_idx`

Execution Time: **0.023 ms** — back to instant !

Fix 2 — Expression Index

```
CREATE INDEX ON erp.payments (
    date_trunc('m', tstamp AT TIME ZONE 'UTC+1'));
```

- **Beware:** this is a LOSSY index
- Only matches queries with the **exact same expression and casts**
- `date_trunc('s', ...)` won't match an `'m'` index
- Preferred: rewrite the predicate to keep the column bare

Selecting and Fetching All the Data

- Developer wants ticket IDs with `status = 10`
- Uses `SELECT *` instead of `SELECT id`
- "I can just discard the columns I don't need"

Selecting and Fetching All the Data

fetch.py results:

- `SELECT * took` **0.245 seconds**
- `SELECT id took` **0.093 seconds**
- ~2.6x slower for just 500 rows, on localhost

fetch2.py — The Horror

```
# Fetches ENTIRE tickets table, filters in Python!
 cur.execute ('''SELECT * FROM support.tickets''')    # 1M+ rows
 res = cur.fetchmany (10000)
 while (res):
      for row in res :
          if row [ 'status' ] == 10 :    # filtering in Python
              tkts += row [ 'id' ],
      res = cur.fetchmany ( 10000 )
```

Result: 0:06:41 to do what SQL does in < 1 second.

Use the database for data retrieval! 30 years of optimization.

The Rules for selecting from the app

- **Name your columns** : `SELECT id, status` not `SELECT *`
- **Filter in SQL** : `WHERE status = 10` , not in Python
- **Paginate** : `LIMIT` + `OFFSET` or keyset pagination
- Fewer selected columns = potential **Index-Only Scans** = much faster

Not Using CTEs — The Scenario

- Find emails of customers notified about unpaid invoices for **services**
- Non-CTE: multiple joins + nested subqueries
- Hard to read, hard for the optimizer

The CTE Rewrite

```
WITH
 unp AS (
      SELECT id , customer c , order_group AS og
      FROM erp.invoices WHERE paid = false ),
 ni AS (
      SELECT og.id FROM erp.order_groups og
      JOIN erp.orders o ON o.order_group = og.id
      WHERE o.item IS NULL )
 SELECT DISTINCT email
 FROM erp.customer_contact_details ccd
 JOIN unp ON unp.c = ccd.id
 JOIN ni ON ni.id = unp.og
 JOIN erp.sent_emails se ON se.invoice = unp.id
 AND se.email_type = 'Invoice reminder' ;
```

CTE & Identifier Tips

CTEs read top-to-bottom like a narrative:

- Since PG 12, CTEs are **inlined** by the optimizer
- Use `MATERIALIZED` to force early evaluation if needed
- Don't name CTEs after existing tables !

Uppercase identifier trap:

- `CREATE TABLE "Invoices"` forces quoting **everywhere, forever**
- `TABLE Invoices;` → ERROR: relation "invoices" does not exist
- Convention: **lowercase _snake _case** , never quote
- Use column aliases for fancy report names instead

Core Takeaways — Correctness

- NOT IN with NULLs → `NOT EXISTS` or `LEFT JOIN / IS NULL`
- BETWEEN on timestamps → `>= start AND < end`
- Integer division → Cast to `::float` or `::numeric`
- COUNT(col) skips NULLs → Use `COUNT(*)` for row counts
- NULL in unique key → `NULLS NOT DISTINCT` (PG 15+)

Core Takeaways — Performance & Structure

- Function on indexed col → Keep the column bare
- SELECT * / fetch all → Name columns, filter in SQL
- Nested subqueries → Use CTEs
- Uppercase identifiers → `lowercase_snake_case`

The Mindset Shift

- "It runs without errors" does **not** mean it's correct
- Test with NULLs, with edge-case timestamps, with large data
- Read `EXPLAIN ANALYZE` — it shows what Postgres **actually** did
- Use the database for what it's good at: 30 years of optimization
- SQL is **not** Python — remember three-valued logic

Resources

- **Book:** manning.com/books/postgresql-mistakes-and-how-to-avoid-them
- **Code:** github.com/vyruss/postgresql-mistakes
- Setup: **`schema.sql` → `customer_dump.sql` → `create_data.sql`**

Thank you!

Discount code **au35ang** for **35% off**