LISTEN Carefully: How NOTIFY Can Trip Up Your Database

Jimmy Angelakos
Staff Software Engineer, pgEdge

PG DATA
Chicago, 2026-06-05

pgEdge®

About Jimmy Angelakos

- Staff Software Engineer, pgEdge. Based in Edinburgh, Scotland
- PostgreSQL Significant Contributor
- Organizer, PostgreSQL Edinburgh User Group
- Member, PostgreSQL Europe Diversity Committee
- Author, PostgreSQL Mistakes and How to Avoid Them
- Co-author, PostgreSQL 16 Administration Cookbook
- Creator of pg_statviz PostgreSQL extension

What is this talk about?

- A feature that's
  - Elegant
  - Powerful
  - Has a trap
- How we stepped in the trap
- How we found out
- How we fixed it

A wonderful feature

- PostgreSQL's `LISTEN/NOTIFY`
- Use case: Async notifications to application
  - Can use for inter-process communication (IPC)
- Built-in, lightweight, zero extra infrastructure
  - No Redis/Kafka/RabbitMQ needed, just SQL

How does it work?

- Simple IPC inside PostgreSQL
- Publisher/Subscriber pattern
- Payload strings up to 8000 bytes
- Transaction-safe
  - Notifications only delivered if the transaction commits
- Sounds awesome, right?

Enough talk. How do I do it?

Usage

- Session A: "I'm listening on this channel"

```
LISTEN order_watchers_json;
```

- Session B: "Tell this channel that something happened"

```
NOTIFY order_watchers_json, '{"order_id": 42,
                              "status": "completed"}';
```

- Session A: Receives from the channel:
  - Asynchronous notification `"order_watchers_json"` with payload `'{"order_id": 42, "status": "completed"}'`

Fun tales from Production

- Architecture:
  Postgres trigger → Legacy app → Redis → Newer components
- `NOTIFY` was used to tell the legacy app that a transaction was committed
- Notable: Significant increase in customer traffic since beginning

Symptoms

- Sudden spike in `canceling statement due to lock timeout`
- Lock cascade, transactions stalling at `COMMIT` time
  - Errors started appearing on busy DB servers
  - Client requests saw increased wait times
- It was necessary to take action
  - Environments with high transaction volume were suffering

Pull the logs!

- Logs showed:

```
LOG: process 285654 acquired AccessExclusiveLock on object
0 of class 1262 of database 0 after 1120.028 ms

LOG: process 286113 acquired AccessExclusiveLock on object
0 of class 1262 of database 0 after 1116.867 ms
```

- Processes were waiting **over 1.1 seconds** just to get a lock!
- But… which lock?

Hunting class 1262

- This wasn't a row lock
- This wasn't a table lock
- `class 1262` is the `pg_database` system catalog
- Wait what?
- Why are we locking the global database catalog for every transaction?!

The trap: serialization

- To guarantee ordering, every single `COMMIT` with a pending `NOTIFY` requires an `AccessExclusive` lock on this global object
- Guaranteeing ordering: Serialization
- By putting `NOTIFY` in triggers, we were forcing concurrent transactions to queue one behind another

"Accidental serialization"

- Treated `LISTEN/NOTIFY` like a concurrent message bus
- Postgres treats it like a globally serialized queue
- Every transaction commit triggered a notification
- As concurrency increased (more customers) →
  more commits competed for the same global lock
- Throughput was collapsing
- Errors started appearing when the DB killed transactions
  that waited too long

Fixing it

Exploring solutions

- First instinct: application-side solution
  - Legacy app, nobody wanted the risk
    - Basically nobody wanted to touch it
  - Solution too complicated
- Can we solve it from the database side?
  - Problem was: notification sending was coupled to the transaction trigger (waited on `COMMIT`)

The fix

Solved it entirely from the database side:

1) Trigger `INSERT`s to an `UNLOGGED` notification **queue table**

   → no global locks

2) The `NOTIFY` function tries to grab an **advisory lock**

   → to ensure there is only one sender at a time

3) Queued notifications are **batch sent**

   → the queue is drained by the lock "winner", "losers" commit instantly

   → One lock acquisition covers the entire batch of `pg_notify()` calls

The notification queue table

- Created an `UNLOGGED` table:
  - Temporary storage of notifications
  - Does not bloat the WAL

```
CREATE UNLOGGED TABLE order.notification_queue (
    tstamp      TIMESTAMPTZ DEFAULT clock_timestamp(),
    payload     TEXT        NOT NULL
);
```

Updating the trigger (queuing)

- We no longer call `pg_notify()` directly
- We `INSERT` the JSON payload into the queue

```
payload := json_build_object(
        'id', NEW.id,
        'action', NEW.action_type,
        'item', NEW.item
    )::TEXT;
IF payload IS NOT NULL THEN
    INSERT INTO order.notification_queue (payload)
    VALUES (payload);
END IF;
```

Updating the trigger (advisory lock)

- After queuing the notification, we attempt to grab an advisory lock
  - Transaction-level
  - Non-blocking

```
-- Attempt to acquire advisory lock (scope: this transaction)
lock_acquired := pg_try_advisory_xact_lock(ADVISORY_LOCK_ID);

-- If we have the lock send notifications, else do nothing
IF lock_acquired THEN
    newest := clock_timestamp();
    cutoff := newest - INTERVAL '10 seconds';
    …
```

Updating the trigger (batching)

- The transaction that obtained the lock fetches the entire queue to batch

```
…
FOR notif IN
    WITH notifs AS (
        DELETE FROM order.notification_queue
        WHERE tstamp <= newest
        RETURNING *
    )
    SELECT n.payload, n.tstamp
    FROM notifs n
    WHERE n.tstamp > cutoff
    ORDER BY n.tstamp
    …
```

Updating the trigger (sending)

- We send all queued notifications in a batch

```
            …
            LOOP
                PERFORM pg_notify('order_watchers_json',
                                  notif.payload);
            END LOOP;
        END IF;

        RETURN new;
    END;
```

Why the fix works

- The advisory lock is a non-blocking try-lock
  - Transactions that fail to get it never wait: they commit immediately
- Only the winner of the advisory lock calls `pg_notify()`
  - So it alone takes the global `NOTIFY` lock – once, for the entire batch
- In practice, `pg_notify()` is barely contended
  - Contention stops scaling: more transactions just means a bigger batch

Testing (trust nothing)

- Correctness
- Concurrency (race conditions)
- Performance
- Regression
- Clone of Production DB with synchronous standby
  - To introduce realistic commit latency
- Stress test: 4x the production transaction volume
  - Result: sustained the rate flawlessly on a smaller instance

Logging (trust nothing) 😏

- Right after the INSERT statement, report the queue length

```
-- Report queue length in ~0.5% of calls for logging
-- 199 is prime so it's unlikely we'll end up in a bad pattern
IF txid_current() % 199 = 0 THEN
    SET log_error_verbosity='terse';
    RAISE WARNING 'order.notification_queue length: %',
pg_stat_get_live_tuples('order.notification_queue'::REGCLASS);
    RESET log_error_verbosity;
END IF;
```

Results

- **Immediate difference**
- Decoupled notifications, truly async: Transactions unblocked immediately
- Lock timeout errors disappeared
- In stress tests, transaction throughput more than **tripled**
- Performance is now bound by hardware again
  - Not by self-induced bottlenecks
- Client requests see sharply reduced wait times

Recap

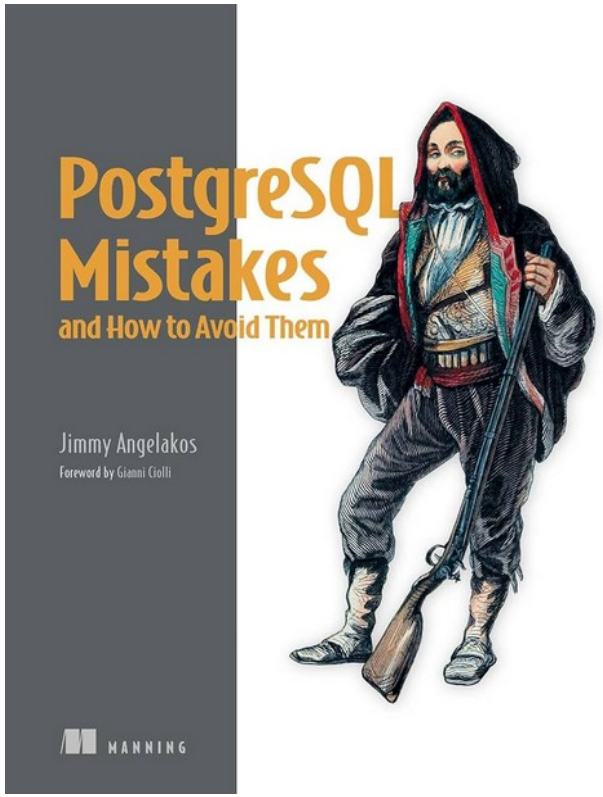
- `LISTEN/NOTIFY` is powerful and elegant
- **However**, it can become a silent bottleneck in high-throughput systems
- Watch out for global locks in high-frequency triggers
- Advisory locks are an awesome tool for concurrency control
- Keep `NOTIFY` **out of your hot path**

Please leave feedback

LISTEN Carefully: How NOTIFY Can Trip Up Your Database

Thank you!

Discount code **pgdata26** for **45% off** all products (valid thru June 19th)



TQRCG