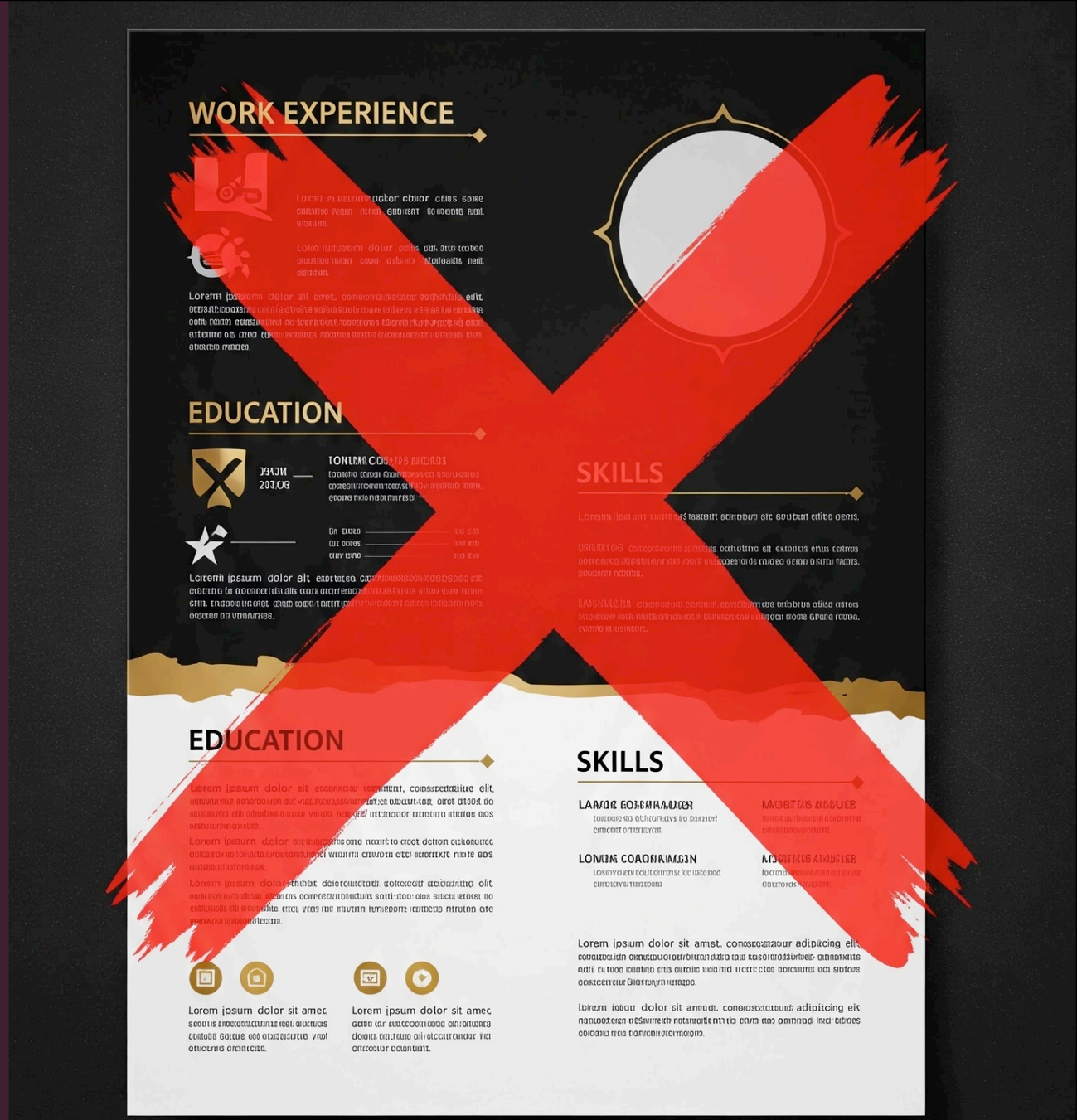
Simple things in PostgreSQL That will break your application

Radim Marek

radim@boringsql.com

boringSQL

The anti-credentials



boringSQL

***Databases* are elegant,**
logical and beautiful systems

boringSQL

... and then we let developers connect to them.

boringSQL

Silly thing developers say about databases

1. It’s just a dumb storage layer.
2. ORMs handle all the SQL so we don’t need to understand it.
3. We should store it as JSON.
4. Let's avoid JOINs for performance

The Confession: I was one of them.

boringSQL

The Graveyard of Shiny Things

```
CORBA, DCOM, RMI-heavy apps, J2EE, JSF, Struts, Aspects, ASP.NET Web Forms,
Rails “classic” apps, Flash, jQuery, NoSQL, Silverlight, ActiveX , Applets,
GWT (Google Web Toolkit), XML-heavy storage, SOAP-heavy web services,
Message brokers (TIBCO, MQ-series)
```

boringSQL

boringSQL

boringSQL

The Disconnect

My Database

- ✓ Thousands rows
- ✓ Single User (You)
- ✓ Most likely Apple Sillicon
- ✓ "Works in 10ms"

You write code.

You test it.

It works.

You ship it.

You go for lunch...

Production Database(s)

- × Millions - billions of rows
- × 2,000+ QPS
- × Stream replication to 10+ replicas (AZs?)
- × Backup jobs running
- × Analytics/Monitoring queries running
- × Analytics/Monitoring queries running
- × The laws of physics apply

"It works in staging" is not a benchmark. It's a comforting lie.

boringSQL

The comforting things we tell to ourselves

```
#1: "SELECT is read-only and safe"
Reality: SELECT holds snapshots that block cleanup

#2: "Transactions protect my data"
Reality: Long transactions are production killers

#3: "Adding an index makes queries faster"
Reality: Indexes have hidden maintenance costs
```

These aren't wrong, exactly.

They're just... incomplete truths.

boringSQL

MVCC

Reality: PostgreSQL uses MVCC

(Multi-Version Concurrency Control)

Every transaction gets a snapshot:

- Shows database as it was when transaction started
- Prevents you from seeing other transactions' changes
- Keeps old row versions alive until snapshot is done

Today's Journey - The Cascading Failure

boringSQL

ACT I - The Integration That Started It All

boringSQL

HubSpot Integration

```ruby
def sync_contact_from_hubspot(hubspot_contact_id)
  ActiveRecord::Base.transaction do
    contact = Contact.find_by(hubspot_id: hubspot_contact_id)

    # Fetch latest data from HubSpot API
    hubspot_data = HubspotClient.get_contact(hubspot_contact_id)

    # Enrich with Clearbit data
    clearbit_data = ClearbitClient.enrich(hubspot_data[:email])

    # Update our database
    contact.update!(
      name: hubspot_data[:name],
      email: hubspot_data[:email],
      company: clearbit_data[:company],
      title: clearbit_data[:title],
      last_synced_at: Time.now
    )

    # Log to audit system
    AuditLogger.log_sync(contact, hubspot_data)
  end
end
```

Friday Afternoon - The Marketing Import

Monday-Thursday: Everything fine

- 5 contacts sync per minute
- 3 seconds per contact = ~15 seconds of work per minute
- Life is good ✓

Friday, 3:00 PM: Marketing does their job

- Import 50,000 updated contacts into HubSpot
- "Let's get these into our system before end of day!"
- Totally reasonable business request

BOOM! You start the investigation...

boringSQL

Boom - idle in transaction

```
SELECT pid, usename, state,
       now() - xact_start as duration,
       left(query, 80) as query
FROM pg_stat_activity
WHERE state = 'idle in transaction'
ORDER BY duration DESC;

 pid   | state                | duration  | query
-------+----------------------+-----------+----------
-----------------
 12451 | idle in transaction  | 00:03:22  | SELECT *
FROM contacts...
 12452 | idle in transaction  | 00:03:18  | SELECT *
FROM contacts...
 12453 | idle in transaction  | 00:03:15  | SELECT *
FROM contacts...
 12454 | idle in transaction  | 00:02:54  | SELECT *
FROM contacts...

 ...
 (45 rows showing 'idle in transaction')
```

```
SELECT schemaname, relname,
       last_autovacuum,
       n_dead_tup,
       autovacuum_count
FROM pg_stat_user_tables
WHERE relname = 'contacts';

 relname   | last_autovacuum     | n_dead_tup |
autovacuum_count
-----------+---------------------+------------+------
------------
 contacts  | 2025-11-24 17:23:11 | 8500000    | 847
```

boringSQL

The VACUUM Misconception

When you google/ask LLM about bloat, top results are

- "Tune autovacuum_vacuum_scale_factor"
- "Increase autovacuum_max_workers"
- "Adjust vacuum_cost_delay"
- "Set autovacuum more aggressive"

```
ALTER SYSTEM SET
autovacuum_vacuum_scale_factor = 0.05;
ALTER SYSTEM SET autovacuum_max_workers = 6;
ALTER SYSTEM SET autovacuum_vacuum_cost_delay
= 2;
```

boringSQL

Emergency fix

```ruby
def sync_contact_from_hubspot(hubspot_contact_id)
  # Phase 1: Read (no transaction)
  contact = Contact.find_by(hubspot_id: hubspot_contact_id)

  # Phase 2: External I/O (no transaction)
  hubspot_data = HubspotClient.get_contact(hubspot_contact_id)
  clearbit_data = ClearbitClient.enrich(hubspot_data[:email])

  # Phase 3: Write (SHORT transaction)
  ActiveRecord::Base.transaction do
    contact.reload
    contact.update!(...data...)
  end  # 50ms instead of 3 seconds

  # Phase 4: Audit (no transaction)
  AuditLogger.log_sync(contact, hubspot_data)
end
```

boringSQL

First defense layer

```
ALTER SYSTEM SET idle_in_transaction_session_timeout = '2s';
SELECT pg_reload_conf();
```

What this does:

- Any transaction idle >5 seconds → killed automatically
- Catches external I/O in transactions
- Catches forgotten transactions
- Catches framework auto-wrapping

boringSQL

ACT II - Optimization
That Made It Worse

boringSQL

Let's Make The Dashboard Faster

Product Manager: "Great work on Friday! Hey, the deal dashboard is slow when sorting users by last login time. Can we speed that up?"

Developer: "Sure, easy fix - just need an index."

boringSQL

Just Use The Index, Luke

On Monday:

```
CREATE INDEX contacts_by_activity ON contacts(last_activity);
```

✓ Query time: 2 seconds → 50ms

✓ Tests pass

✓ Dashboard is now fast again

SHOOT!$$#$

boringSQL

What Indexes Actually Cost

```
UPDATE contacts SET last_activity = NOW() WHERE id = 123456;
```

Before the index:

1. Write new row version to same heap page → new tuple (HOT update)
2. Mark old row version as dead → dead tuple
3. No index updates needed - all indexes still point to same page
4. Generate minimal WAL (just heap change)
5. Replicate to standbys

After the index

```
UPDATE contacts SET last_activity = NOW() WHERE id = 123456;
```

With the index:

1. Write new row version to heap → new tuple (may need new page)
2. Mark old row version as dead → dead tuple
3. Update contacts_pkey (id) → new pointer
4. Update contacts_by_owner_id → new pointer
5. Update contacts_by_creation → new pointer
6. Update contacts_by_activity → new entry with new value
7. Generate WAL for all these changes (heap + 4 indexes)
8. Replicate to standbys

boringSQL

Wednesday Afternoon -

Monday-Tuesday: Everything fine

```
• Life is good ✓
```

Wednesday, 3:00 PM: Something is off, the app is now sluggish

```
• Dead tuples: 15,000,000 (was 2,000,000 on Monday)
• Bloat ratio: 35% (was 15% on Monday)
• HOT update ratio: 0% (was 95% on Monday)
• VACUUM running constantly (was occasional)
```

boringSQL

```sql
SELECT schemaname, relname, n_tup_upd, n_tup_hot_upd,
       round(100.0 * n_tup_hot_upd / NULLIF(n_tup_upd, 0), 1) as hot_pct
FROM pg_stat_user_tables
WHERE relname = 'contacts';
```

```
 relname  | n_tup_upd | n_tup_hot_upd | hot_pct
----------+-----------+---------------+---------
 contacts | 2500000   | 0             | 0.0
```

boringSQL

But but it's only minor slow down on updates?!? Right?

```
-- The query that was supposed to be FAST (50ms on Monday)
SELECT * FROM contacts
WHERE last_activity > NOW() - INTERVAL '14 days'
ORDER BY last_activity DESC;

-- Now: 1.02 seconds (and getting worse)
-- SeqScan in full swing
```

boringSQL

Index Bloat: What VACUUM Can't Fix

```
CREATE EXTENSION pgstattuple;
SELECT * FROM pgstatindex('contacts_by_activity');

 avg_leaf_density | leaf_fragmentation
------------------+--------------------
      32.4        |        67.8
-- Should be 90%  |  Should be < 10%
```

The only "fast" fix: REINDEX CONCURRENTLY (rebuild from scratch)

boringSQL

This is how production systems degrade

Not one big mistake

A series of reasonable decisions

boringSQL

ACT III: The Best Practice That Killed Everything

boringSQL

Monday 11:00

```
ERROR: canceling statement due to conflict with recovery
DETAIL: User query might have needed to see row versions that must be
removed.
```

🥁🥁🥁 No reports workings... weekly management meeting is about to start

The Fix: Enable hot_standby_feedback

```
-- Set hot_standby_feedback = on on the standby, which
-- passed back information to the primary that certain
-- rows are still needed in a table. This will cause bloat
-- on the primary, but allows queries on the secondary to
-- finish reliably.

ALTER SYSTEM SET hot_standby_feedback = on;
SELECT pg_reload_conf();
```

Analytics team is happy

Business is happy 🎉

boringSQL

Wednesday: The Primary Starts Dying

```
-- On primary, everything looks fine locally
SELECT * FROM pg_stat_activity WHERE state =
'idle in transaction';
-- 0 rows! No long transactions here!

-- But check this:
SELECT * FROM pg_replication_slots;
-- or
SELECT client_addr, backend_xmin,
       age(backend_xmin) as xmin_age
FROM pg_stat_replication;

-- backend_xmin from 6 hours ago...
-- THAT's what's blocking VACUUM
```

oh the irony

Fixed the transaction duration ✓

Added idle_in_transaction_timeout ✓

Set up read replicas to offload analytics ✓

Enabled hot_standby_feedback to prevent query cancellations ✓

boringSQL

Metrics

Metrics:

• Replica CPU: 60% (expected for analytics)

• Primary CPU: 70% (high but not maxed)

• Connections: Primary 150/200, Replica 20/50

• No errors in logs

• Replication lag: <1 second

What's wrong?

A) Replica stealing CPU from primary

B) Network latency between primary and replica

C) Replica queries somehow affecting primary VACUUM

D) Too many connections on primary

E) This is impossible - replicas can't affect primary

The Hidden Connection

Answer: C - Replica queries are blocking primary VACUUM
The mechanism:

1. Analytics query starts on replica
2. Query scans 10 million rows, takes 2 minutes
3. Replica needs a consistent view for those 2 minutes
4. Replica → Primary: "Don't clean up rows I might need"
5. Primary VACUUM: "Okay, I'll wait"
6. For 2 minutes, primary accumulates dead tuples
7. Query finishes
8. But another analytics query starts...
9. And another...
10. Primary VACUUM blocked continuously

What hot_standby_feedback Does

```
WITHOUT hot_standby_feedback (default: OFF):

Replica behavior:
• Long query running (analyzing 2 years of data)
• Primary sends WAL: "I deleted these rows"
• Replica: "But my query needs those rows!"
• After 30 seconds: ERROR: canceling statement due to conflict with recovery
• Query fails ×
• Analytics team complains ×
• But: Primary stays healthy ✓

WITH hot_standby_feedback (ON):

Replica behavior:
• Long query running (analyzing 2 years of data)
• Replica → Primary: "Hold XID 5,234,891, don't clean yet"
• Primary: "Okay, I won't clean those rows"
• Query runs for 2 minutes successfully ✓
• Analytics team happy ✓
• But: Primary accumulates dead tuples for 2 minutes ×
```

The Timeline of the bloat

```
Primary sees:
6:00-6:02 PM: Daily report holds XID 5,234,891
              VACUUM: "Waiting..."
              User traffic: 1,000 UPDATEs/sec × 120 sec = 120,000 dead
tuples

6:02-6:07 PM: Weekly aggregation holds XID 5,235,200
              VACUUM: "Still waiting..."
              User traffic: 1,000 UPDATEs/sec × 300 sec = 300,000 dead
tuples

6:07-6:15 PM: Monthly rollup holds XID 5,236,100
              VACUUM: "STILL waiting..."
              User traffic: 1,000 UPDATEs/sec × 480 sec = 480,000 dead
tuples
```

The smoking gun

```
SELECT client_addr,
       state,
       backend_xmin,
       replay_lag
FROM pg_stat_replication;

  client_addr     |   state     | backend_xmin | replay_lag
------------------+-------------+--------------+------------
  10.0.1.52       | streaming | 5234891      | 00:00:00.2
```

boringSQL

Meanwhile on replica for analytics

```
SELECT pid, state,
       now() - xact_start as duration,
       left(query, 80) as query
FROM pg_stat_activity
WHERE backend_xmin IS NOT NULL;


 duration  |  query
-----------+--------
 01:03:15  | SELECT date_trunc('month', created_at), count(*),
sum(revenue)...
```

boringSQL

#incidents -
147 messages

[CEO has joined the channel]

"Everything was fine three weeks ago. What changed?"

Three things changed.

And they compounded.

boringSQL

Number of simple things

Big Compound disaster

boringSQL

The emergency fix

1. Sacrifice the reporting on day 1 (give ETA)
2. Manually trigger VACUUM (not FULL) on the worst affected table (can be dozens at this point)
3. Watch the progress

```
SELECT schemaname, relname, n_dead_tup, last_autovacuum
FROM pg_stat_user_tables
WHERE n_dead_tup > 10000
ORDER BY n_dead_tup DESC;
```

boringSQL

Fix the HOT updates

PostgreSQL for everything works...

But might not always be a good solution

boringSQL

Setup monitoring

Idle in Transaction

```
SELECT pid, usename, state,
       now() - xact_start as duration,
       left(query, 80) as query
FROM pg_stat_activity
WHERE state = 'idle in transaction'
  AND now() - xact_start > interval '1 second'
ORDER BY duration DESC;
```

Dead Tuple Accumulation

```
SELECT schemaname, relname,
       n_dead_tup, n_live_tup,
       round(100.0 * n_dead_tup /
NULLIF(n_live_tup, 0), 1) AS
dead_pct,
       last_autovacuum
FROM pg_stat_user_tables
WHERE n_dead_tup > 10000
ORDER BY n_dead_tup DESC;
```

HOT Update Ratio

```
SELECT schemaname, relname,
       n_tup_upd, n_tup_hot_upd,
       round(100.0 * n_tup_hot_upd /
NULLIF(n_tup_upd, 0), 1) AS hot_pct
FROM pg_stat_user_tables
WHERE n_tup_upd > 1000
ORDER BY n_tup_upd DESC;
```

```
SELECT
    client_addr,
    backend_xmin,
    age(backend_xmin) as xmin_age,
    replay_lag
FROM pg_stat_replication;
```

boringSQL

The Compound Effect

```
Week 1: Transaction duration problem
        Dead tuples/hour: 50,000
        VACUUM capacity: 200,000/hour
        Net: VACUUM keeping up ✓

Week 2: + HOT disabled by index
        Dead tuples/hour: 250,000 (5x increase)
        VACUUM capacity: 200,000/hour
        Net: Slowly losing ground ⚠️

Week 3: + hot_standby_feedback blocking VACUUM
        Dead tuples/hour: 250,000
        VACUUM capacity: 0/hour (during analytics window)
        Net: 6 hours of zero cleanup = 1.5M dead tuples
             Every. Single. Day.
```

boringSQL

The Real Lesson

This wasn't a PostgreSQL problem.

This was a **mental model** problem.

```
We thought: "Transactions protect data"
Reality:    Transactions hold snapshots that block cleanup

We thought: "Indexes make queries faster"
Reality:    Indexes disable optimizations and multiply maintenance

We thought: "Replicas isolate workloads"
Reality:    Replicas can reach back and choke the primary
```

boringSQL

The (Not So) Beautiful Truth

```
PostgreSQL tells you the truth.
But it tells you in pieces.

Each piece is correct.
But the whole picture?
That only emerges in production.
At scale.
At any time.
When someone's pager goes off.
```

boringSQL

What I Want You To Remember

Not the queries.

Not the settings.

Not the emergency fixes.

One thing:
Every operation in PostgreSQL has a cost you don't see until scale reveals it.

```
SELECT holds snapshots.
Transactions block cleanup.
Indexes multiply writes.
Replicas can poison primaries.
```

boringSQL

Now Let's Make It Actionable

```
pg_stat_activity      → Who's holding snapshots?
pg_stat_user_tables   → Where's the bloat?
pg_stat_replication   → Is my replica choking my primary?
pgstattuple           → Are my indexes healthy?
```

boringSQL

Simple Thing No 1 - Transactions

The Rules

1. Never do external I/O inside a transaction
2. Measure transaction duration, not query duration
3. Set a hard limit, be relentless

```
-- Transaction / snapshot issues
statement_timeout
idle_in_transaction_session_timeout
lock_timeout

-- Replication
max_standby_streaming_delay
max_standby_archive_delay
hot_standby_feedback
vacuum_defer_cleanup_age
```

boringSQL

It's not just global settings

```
idle_in_transaction_session_timeout = '5s';
statement_timeout = '10s';
lock_timeout = '3s';

ALTER ROLE app_user SET statement_timeout = '5s';
ALTER ROLE app_user SET idle_in_transaction_session_timeout = '1s';

ALTER ROLE migration_user SET lock_timeout = '60s';
ALTER ROLE migration_user SET statement_timeout = '30min';
```

boringSQL

Transaction for developers

Still one of tho most misunderstand concepts

Transaction = Breath hold

The database holds its breath until you commit.

Don't make it hold its breath while you call an API.

Short transaction = Happy database

boringSQL

But What About 2PC/XA?

❌ Two-Phase Commit: The "solution" that doubles your problems.

```
2PC = 2X Problems
    = 2X Latency
    = 2X Failure modes
    = 2X Complexity
    = 2X The reason your DBAs/DBREs drinks
```

❌ Long transaction hoping nothing fails

❌ 2PC/XA hoping databases coordinate

✅ Short transactions (database writes only)

✅ Idempotency keys (safe retries)

✅ Outbox pattern (reliable events)

✅ Compensation (undo what you can't prevent)

boringSQL

IT (all) DEPENDS

boringSQL

Philosophy of boringSQL

boringSQL