The Postgres Mistakes We All Still Make

Scott Mead
Senior Staff Engineer · Lakebase · Databricks

NYC Postgres Meetup · November 18, 2026

The Premise

- Postgres is thirty years old.
- I've been working on it for 22.
- We're all doing the same [bad] things over and over.

- Every mistake here "works on my laptop"
- Every one shows up in metrics
- They all seem innocent at first

Mistake #0: it's always connections

Postgres runs a process per connection.

- **connect** is `fork()`, memory, authentication, a cold catalog cache, before any row is touched
- **a 5 ms query** connect:~2ms run: 5 ms
  - Without a pool, every request is a connect / disconnect
- **rate** is the issue. A number of connections is one problem, a high rate of connections grinds the system

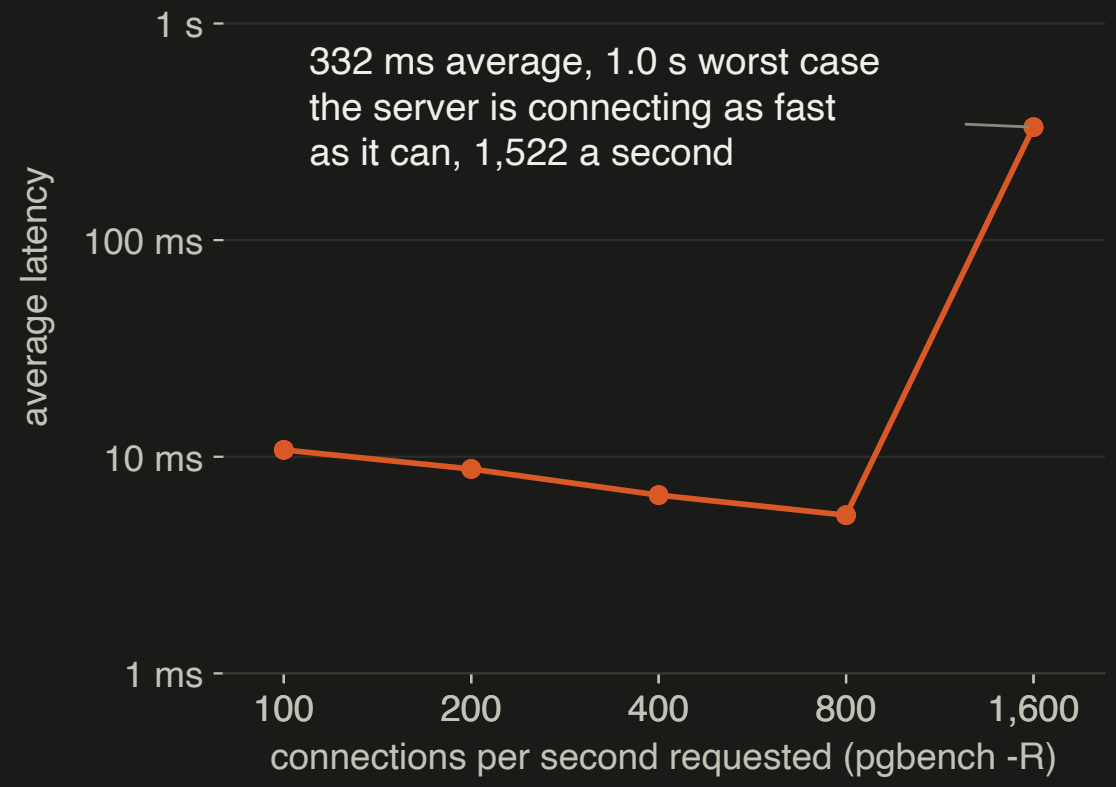
A pool connects once and reuses the backend

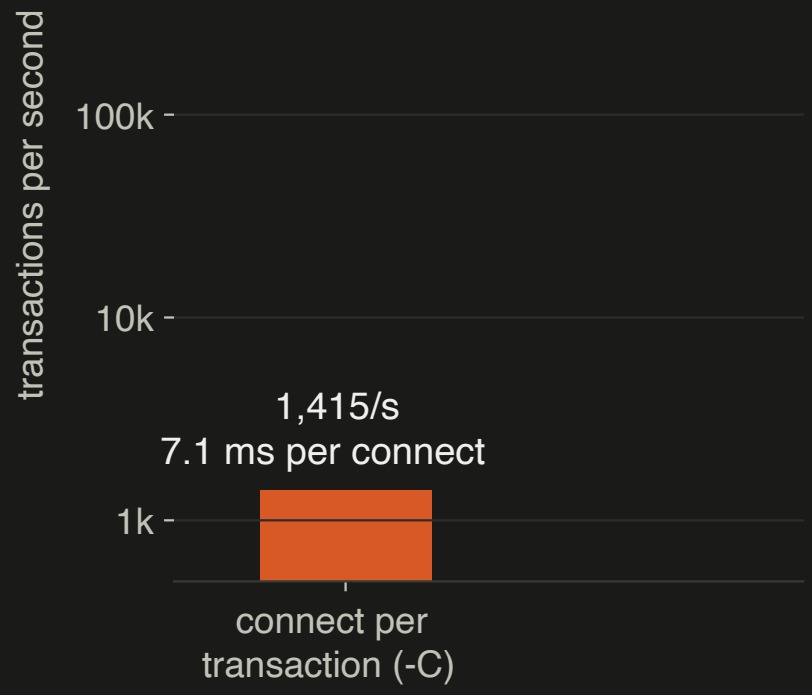
Where the rate comes from

```
def handle(request):
    conn = psycopg2.connect(DSN)        # a new connection per request, no pool
    try:
        with conn.cursor() as cur:
            cur.execute("SELECT ... WHERE id = %s", (request.user_id,))
            return cur.fetchone()
    finally:
        conn.close()                    # torn down again. Next request: fork again
```

- Or forty app servers, each with a pool of fifty per core
- Or reconnect with no delay

pgbench, 20 clients, local PG 17: 1,415/s connecting per transaction





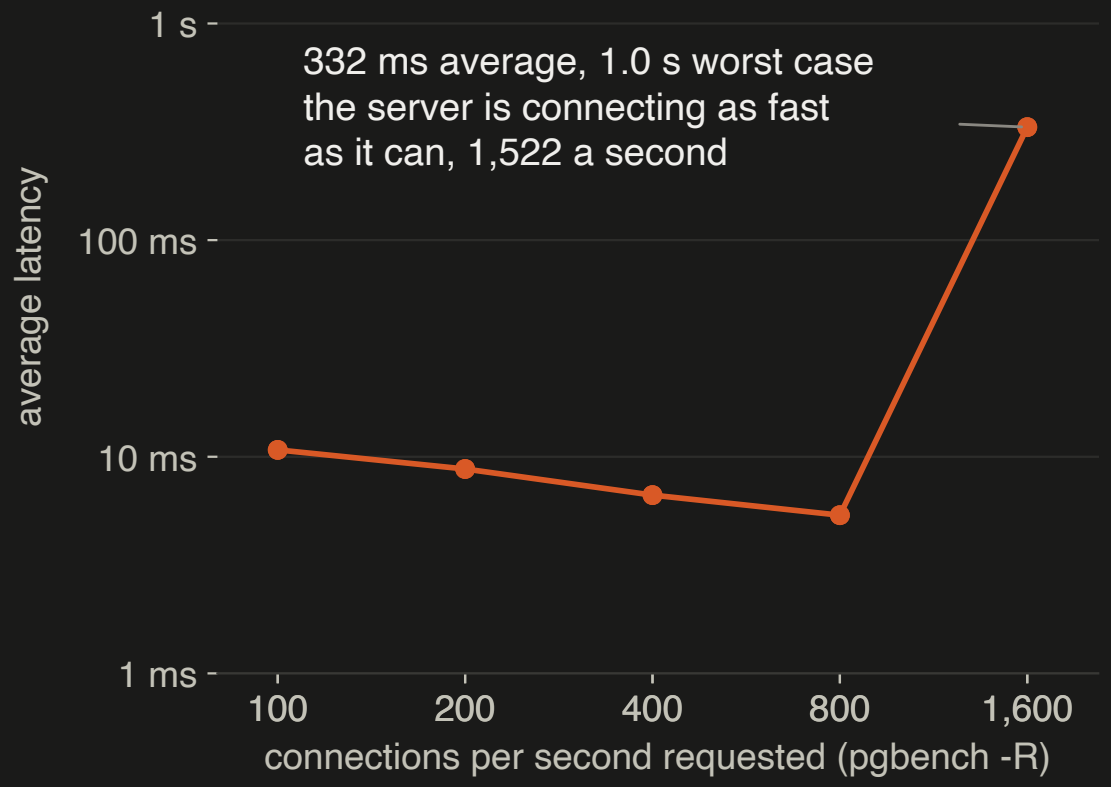
PostgreSQL 17.10, 10-core laptop, unix sockets, select-only, scale 10 (1,000,000 rows).

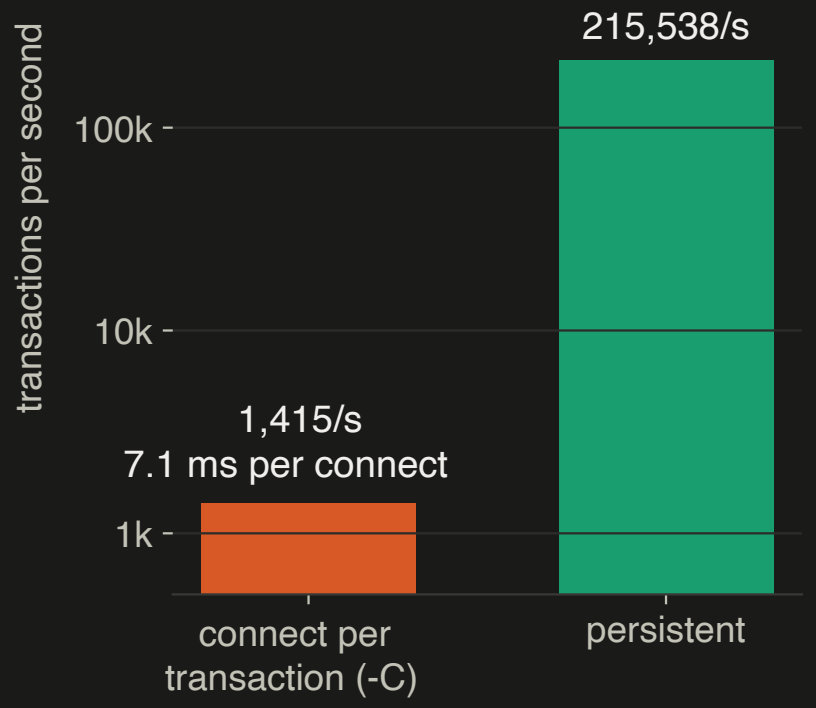
Use a pooled connection

```python
from psycopg2 import pool

db = pool.ThreadedConnectionPool(minconn=2, maxconn=10, dsn=DSN)   # once, at startup

def handle(request):
    conn = db.getconn()                # borrow an open connection, no fork
    try:
        with conn.cursor() as cur:
            cur.execute("SELECT ... WHERE id = %s", (request.user_id,))
            return cur.fetchone()
    finally:
        db.putconn(conn)               # give it back, still connected
```

Same pattern in psycopg 3, HikariCP in Java, and `database/sql` in Go

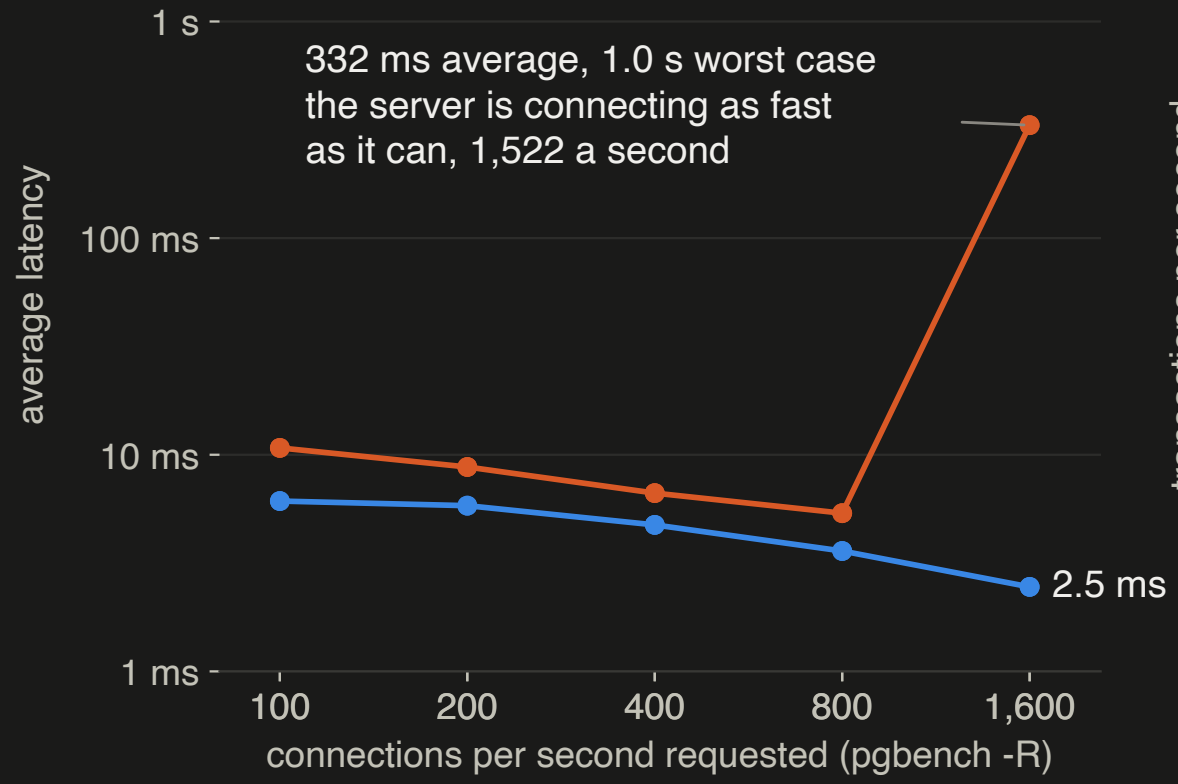
pgbench, 20 clients, local PG 17: 1,415/s connecting per transaction, 215,538/s persistent

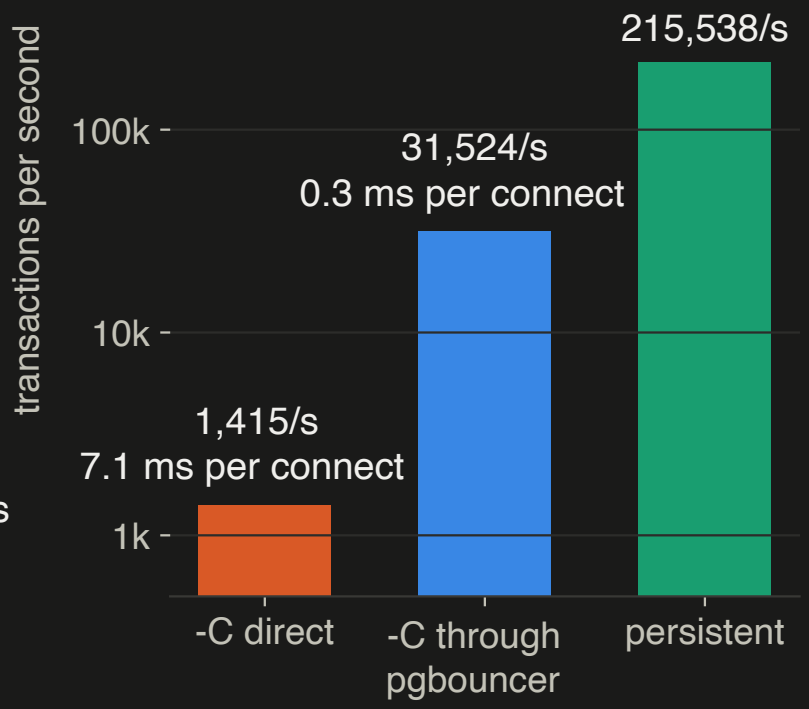




PostgreSQL 17.10, 10-core laptop, unix sockets, select-only, scale 10 (1,000,000 rows).

pgbench, 20 clients, local PG 17: 1,415/s -C direct, 31,524/s -C through pgbouncer, 215,538/s persistent





PostgreSQL 17.10, 10-core laptop, unix sockets, select-only, scale 10 (1,000,000 rows). pgbouncer 1.26, transaction mode, pool 20.

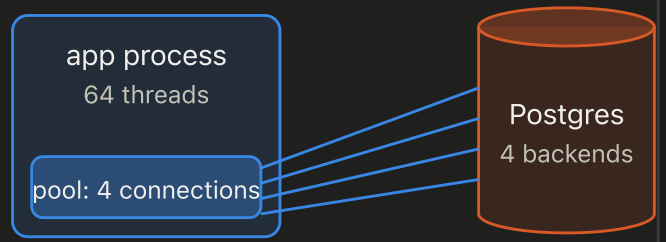
Detection

```
-- PG 14+. sample twice, a minute apart; the difference is the churn
SELECT now(), datname, sessions,
       sessions_abandoned, sessions_fatal,
       sessions_killed, stats_reset
  FROM pg_stat_database
```

- `sessions` : sessions established on this database since `stats_reset`
- `sessions_abandoned` : the client went away without disconnecting
- `sessions_fatal` : ended by a FATAL error
- `sessions_killed` : ended by an operator

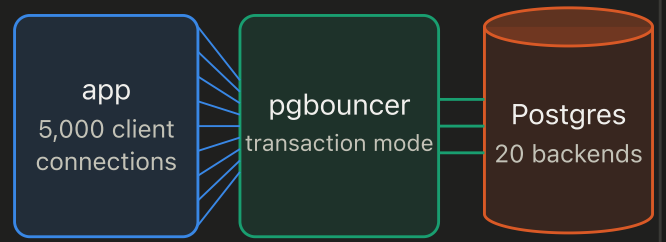
Deployment Options

1. a pool in the application



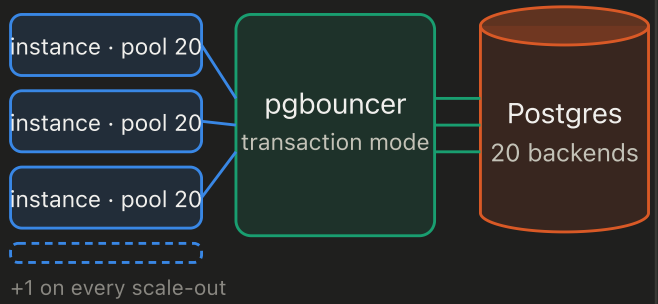
sized to the cores, not to the thread count

2. pgbouncer in transaction mode



5,000 clients in, 20 server connections out

3. both, when the app autoscales



40 × 20 = 800 clients in, still 20 backends

Mistake #1: ignoring MVCC

1. Bulk load

step 1 of 11 back (b) next (n) play (a)

users, heap page 0 8,192 bytes, 56 free

xmin is the xid that wrote the version. xmax is the xid that deleted or replaced it

xmin	xmax	id	first_name	last_name	
500		1	Jane	Smith	48 bytes
500		2	Mike	Rivera	48 bytes

and the other 150 rows on this page

xid 500 wrote both rows and committed. 56 bytes left on the page

free space map (FSM), one entry per page
page 0: 56 bytes free

xid 500 bulk-loads the table in one transaction and commits. The page is otherwise full

The `users` table in 2026

```
CREATE TABLE users (
                id bigint PRIMARY KEY,
                profile jsonb );
```

users, heap page 0 — 8,192 bytes, two rows, 221 bytes used

xmin	xmax	id	profile	bytes
500		1	{"id": 1, "name": "Jane Smith", "email": "jane@example.com"}	110
500		2	{"id": 2, "name": "Mike Rivera", "email": "mike@example.com"}	111

xid 500 wrote both rows and committed. Each profile is one jsonb value, 78 and 79 bytes, plus a 32-byte tuple header and id

One year in

```
-- row 1, version 2, after a year of features: 688 bytes
{"id": 1, "name": "Jane Smith-Rivera", "email": "jane@example.com",
 "spouse": {"id": 2, "name": "Mike Rivera"},
 "addresses": [{"type": "home", "line1": "14 Bryant Park Pl",
                "city": "New York", "zip": "10018"},
               {"type": "work", "line1": "350 Fifth Ave",
                "city": "New York", "zip": "10118"}],
 "preferences": {"theme": "dark", "language": "en-US",
                 "notifications": {"email": true, "sms": false, "push": true}},
 "login_history": ["2026-11-01T08:12:00Z", "2026-11-03T18:40:11Z",
                   "2026-11-09T07:55:43Z", "2026-11-14T21:03:27Z",
                   "2026-11-17T06:48:09Z"],
 "updated_at": "2026-11-18T18:30:00Z"}
```

- Version 1 of row 1 is dead. Its 110 bytes fit nothing in this table any more
- `profile || '{"theme": "light"}'` rewrites all 688 bytes, not the 17 you changed
- One login a day in `login_history`: 250 KB of dead versions per user per year

A year of logins on one row

users, Jane's row, one login a day — 221 pages

xmin	xmax	id	profile	bytes	
500	502	1	{"name": "Jane Smith", ...}	110	dead until VACUUM
502	610	1	{..., "login_history": [1 login]}	592	dead until VACUUM
610	731	1	{..., "login_history": [2 logins]}	616	dead until VACUUM
731	858	1	{..., "login_history": [3 logins]}	640	dead until VACUUM
361 more versions, each 24 bytes bigger than the last					
46,210		1	{..., "login_history": [365 logins]}	9,328	live

one live row, 364 dead versions, 1.8 MB. A reader of this row steps over all of it

with autovacuum keeping up, the free space map after each UPDATE

version 2 freed 592 bytes. Version 3 needs 616. It does not fit, and neither does any version after it

a document that only grows never fits the space its last version freed

arithmetic from the measured sizes

one user, one year
1.8 MB, 221 pages

one million users
1.8 TB of versions

live data in it
9.3 GB

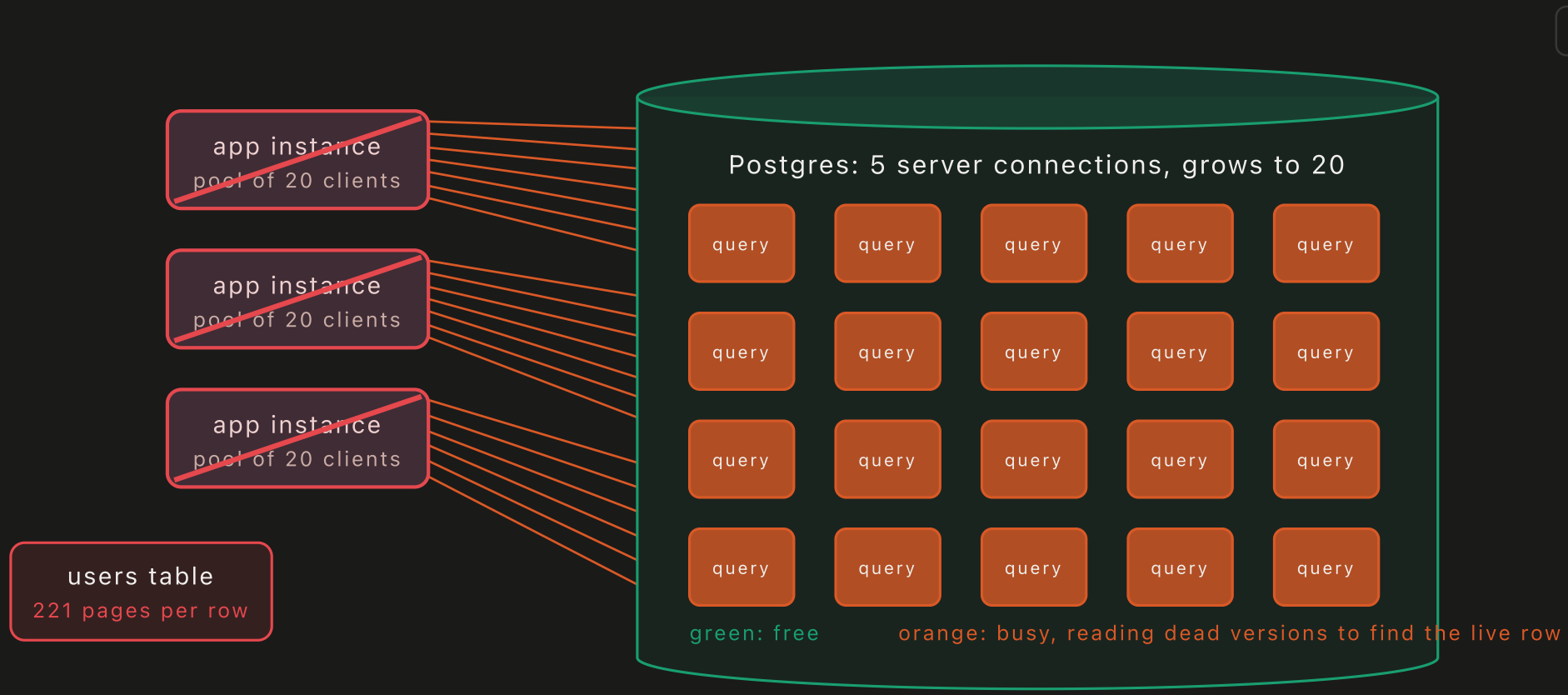
bloat, dead versions
1.80 TB (1,801,072 bytes per user)

the fix
login_history is a table,
not a key in the document.
INSERT 24 bytes, UPDATE nothing

It's always connections

- The reader steps over 364 dead versions to find the live one
- 221 pages where there should be one. The row is a disk read, not a cache hit
- Every query holds its connection for the extra pages. Same traffic, longer holds
- The pool opens more connections to keep up, months after the schema decision
- Autovacuum reads and rewrites the same pages, competing with the queries for the disk

Same traffic, longer holds



221 pages per row. The database is saturated, the app is down, and not a line of code changed

The fix

- The document grows with every login, so every version is bigger than the gap the last one freed
- Facts that accumulate go in a table. The `users` row never changes again

```
-- before: one login rewrites the whole profile, 9,296 bytes by day 365
UPDATE users
   SET profile = jsonb_set(profile, '{login_history}',
                           profile->'login_history' || to_jsonb(now()))
 WHERE id = 1;

-- after: a table. INSERT one 40-byte row, UPDATE nothing
CREATE TABLE login_history (
       user_id       bigint       NOT NULL REFERENCES users (id),
       logged_in_at  timestamptz  NOT NULL DEFAULT now(),
       PRIMARY KEY (user_id, logged_in_at)
);

INSERT INTO login_history (user_id) VALUES (1);
```

Better, not best

- Overwrite a field instead of appending to a list. Every version is the same 110 bytes
- The gap vacuum frees fits the next version, so the table stops growing
- Still one dead version per login. Vacuum has to keep up, and a pinned horizon still bites

```
-- same size every time: replace last_login, keep no history in the document
UPDATE users
   SET profile = jsonb_set(profile, '{last_login}', to_jsonb(now()))
 WHERE id = 1;
```

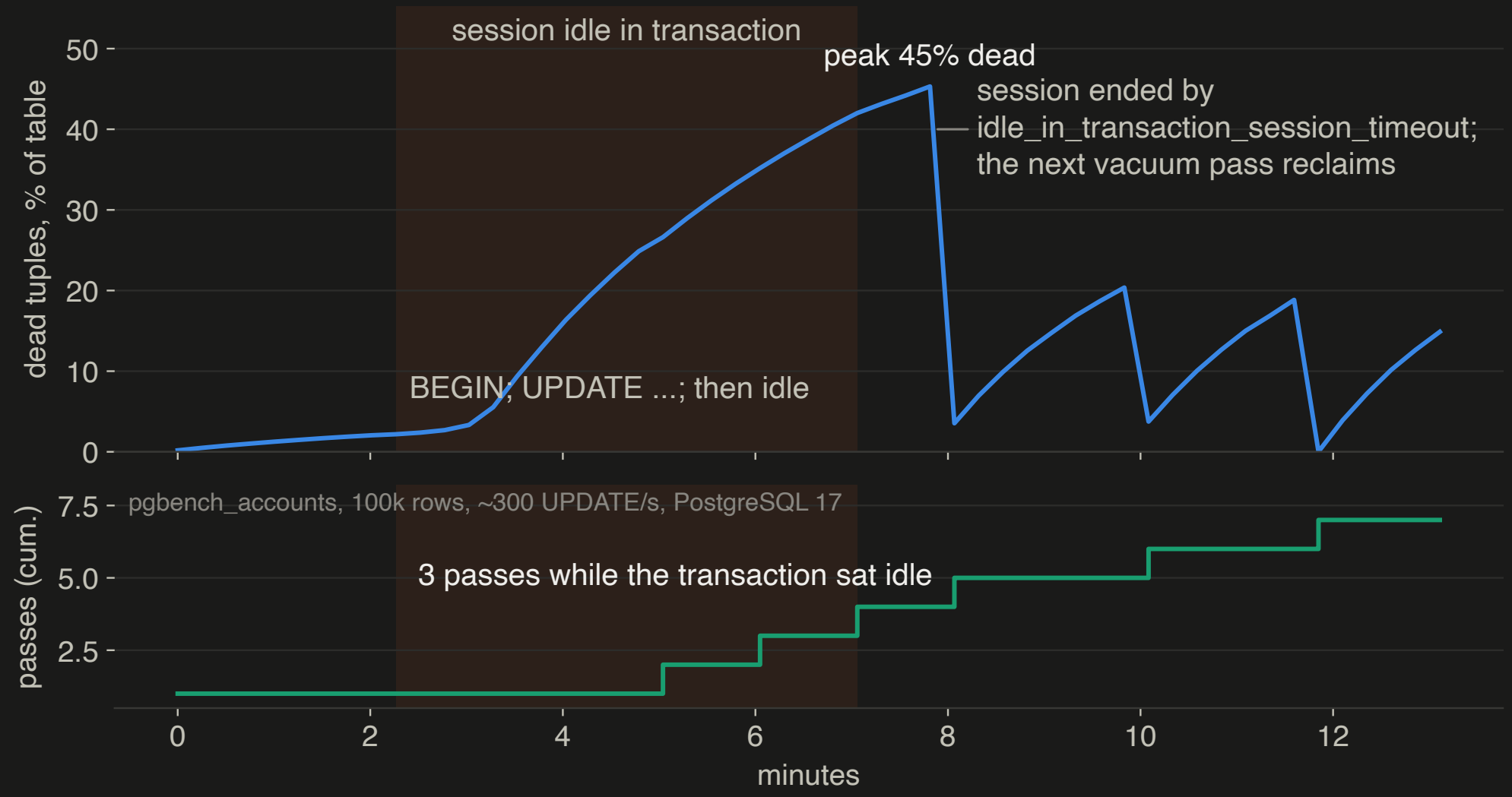
Mistake #2: we leave transactions open

```
cur.execute("SELECT * FROM orders WHERE id = %s FOR UPDATE", (oid,))

resp = requests.post("https://api.payments-provider.com/v1/charges",     # HTTP to a third party
                     json={"card": card, "amount": amount})

cur.execute("UPDATE orders SET status = 'paid' WHERE id = %s", (oid,))
conn.commit()
```

- Autocommit is off*; the SELECT opened a transaction
- It holds the row lock, and it holds a transaction id
- In READ COMMITTED an xid pins the horizon the way a snapshot does

* in psycopg2/3. JDBC, others differ here

One idle transaction: 45% dead, 3 vacuum passes, nothing reclaimed



What vacuum tells you

```
INFO:  finished vacuuming "app.public.orders": index scans: 0
tuples: 0 removed, 1483112 remain, 612004 are dead but not yet removable
removable cutoff: 982144, which was 61930 XIDs old when operation ended
```

"dead but not yet removable" with an old cutoff: someone holds the horizon.

Holder	What it pins	Where to look
Report running 40 minutes *	snapshot (`xmin`)	`pg_stat_activity`
Nightly batch in one transaction	xid and snapshot	`pg_stat_activity`
`pg_dump` at noon	snapshot	`pg_stat_activity`
Physical slot with `hot_standby_feedback`	`xmin` for every table	`pg_replication_slots`
Logical slot nobody consumes	`catalog_xmin` ; WAL is retained	`pg_replication_slots`
Prepared transaction nobody finished	`xmin` , indefinitely	`pg_prepared_xacts`

* A read-only transaction holds one too.

What happens when you can't vacuum

users, Jane's row, last_login updated once an hour — 1 page

xmin	xmax	id	profile	bytes	
500	502	1	{"name": "Jane Smith", ...}	110	dead, cannot be vacuumed
			xmin horizon = 501, one checkout stuck on the gateway for 48 hours, since Fri 23:00		
502	1,000,517	1	{..., "last_login": "Sat 00:00"}	110	dead, cannot be vacuumed
1,000,517	2,000,533	1	{..., "last_login": "Sat 01:00"}	110	dead, cannot be vacuumed
2,000,533	3,000,548	1	{..., "last_login": "Sat 02:00"}	110	dead, cannot be vacuumed
44 more versions, 110 bytes each, a million xids apart					
48,000,611		1	{..., "last_login": "Sun 23:00"}	110	live

one live row, 48 dead versions, 5.4 KB. A reader of this row steps over all of it

the free space map never gets an entry

autovacuum ran all weekend. Cutoff 501 every time, 48 million commits behind, nothing removed

with vacuum unable to progress past an old xid, these rows have no chance of being reused

arithmetic from the measured sizes

one user, one weekend
5.4 KB, 1 page

one million users
5.4 GB of versions

live data in it
110 MB

bloat, dead versions
5.3 GB (5,280 bytes per user)

the fix
COMMIT or kill the stuck session.
The next vacuum pass removes
48 million versions

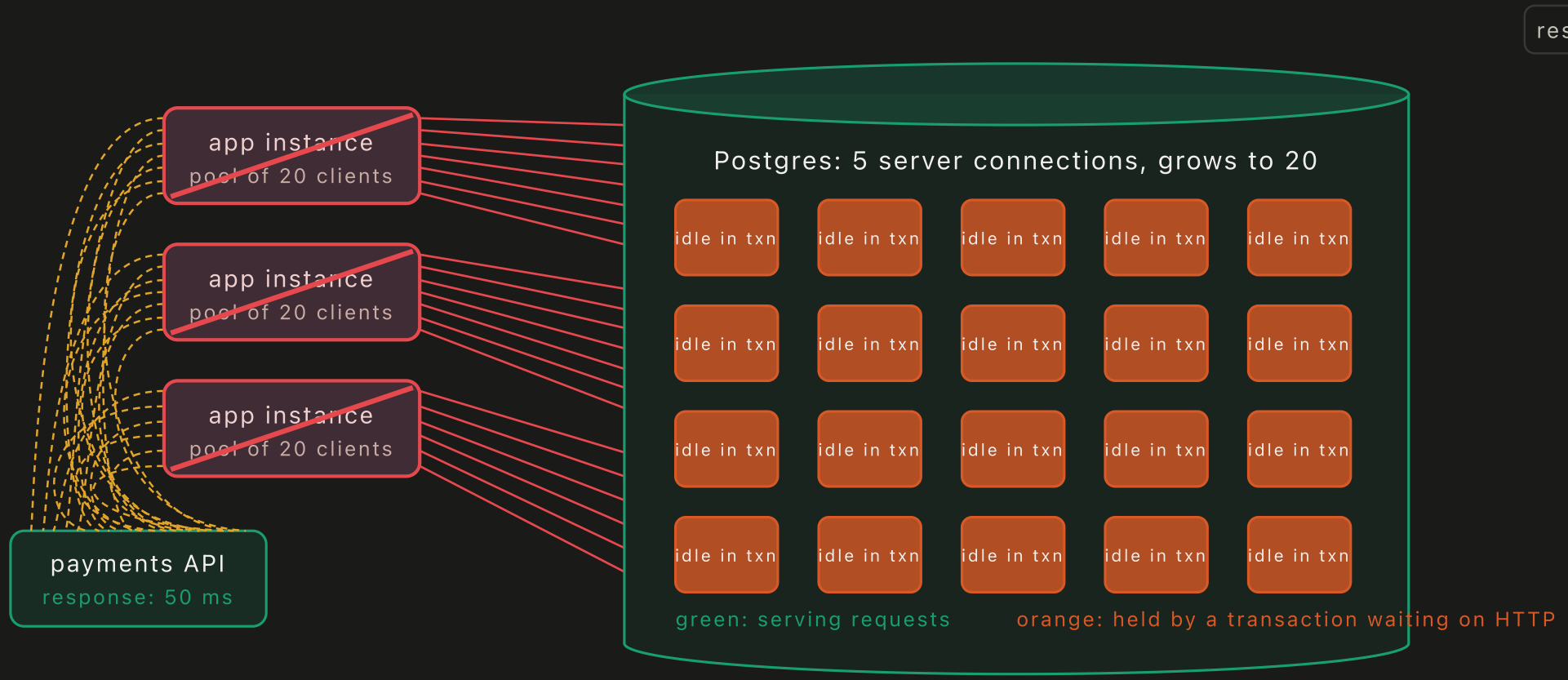
Add a timeout?

```
cur.execute("SELECT * FROM orders WHERE id = %s FOR UPDATE", (oid,))

resp = requests.post("https://api.payments-provider.com/v1/charges",     # HTTP to a third party
                     json={"card": card, "amount": amount}, timeout=30)  # ? to 30 seconds, sometimes never

cur.execute("UPDATE orders SET status = 'paid' WHERE id = %s", (oid,))
conn.commit()
```

- Autocommit is off*; the SELECT opened a transaction
- <= 30 second `idle in transaction`. At 300 commits a second, 9,000 can pile up
- It holds the row lock, and it holds a transaction id
- In READ COMMITTED an xid pins the horizon the way a snapshot does

* in psycopg2/3. JDBC, others differ here

It's always connections



the payments API recovers. Every connection is still held and every app instance is still stuck

Don't ignore MVCC

Detection

```sql
-- backends holding a snapshot or an xid, oldest first
SELECT pid, datname, application_name, state,
       age(backend_xmin) AS xmin_age,
       age(backend_xid) AS xid_age,
       now() - xact_start AS xact_time
  FROM pg_stat_activity
WHERE backend_type = 'client backend'
  AND (backend_xmin IS NOT NULL OR backend_xid IS NOT NULL)
ORDER BY greatest(age(backend_xmin), age(backend_xid)) DESC;
```

- `xmin_age` is how many transactions committed since this one started.
- `xact_time` 10s * 500 xact/sec = 5,000 commits
- Slots and prepared transactions: `pg_replication_slots`, `pg_prepared_xacts`, both in the demo file

The fix

- `FOR UPDATE` plus the charge was never one transaction. Postgres cannot roll back an HTTP call
- Autocommit on. Charge first with an idempotency key, then one guarded `UPDATE`

```
# borrow a connection. The pool was built with autocommit on
conn = pool.getconn()

# the charge runs with no transaction open
resp = requests.post("https://api.payments-provider.com/v1/charges",
                     headers={"Idempotency-Key": f"order-{oid}"},
                     data={"amount": amount, "currency": "usd",
                           "customer": cust, "confirm": "true"}, timeout=30)

# one UPDATE. The status guard replaces the row lock
cur.execute("UPDATE orders SET status = 'paid' "
            "WHERE id = %s AND status = 'pending'", (oid,))

# the only transaction. SQL only, milliseconds
with conn.transaction():
    cur.execute("INSERT INTO ledger ...")
    cur.execute("UPDATE balances ...")
```

The gateway goes down, Postgres goes idle



the payments API recovers. The waiting requests run their UPDATE and finish. Nothing to clean up

The fix, server side

```
ALTER ROLE app_rw SET idle_in_transaction_session_timeout = '30s';
ALTER ROLE app_rw SET transaction_timeout = '5min';      -- PG 17+: the whole transaction, idle or not
```

- Per role: the app and the reports need different numbers
- Pick the seconds from your commit rate. 30 seconds at 300 commits a second is 9,000 transactions held back

alerting today	oldest idle-in-transaction age > 30 s; transactions open > 60 s	yes
	xid age in transactions, the measure that matters; replication slots, prepared transactions	no
verify	rerun the detection query; `xmin_age` falls back to the hundreds	

Mistake #3: bulk DELETE instead of partitioning

Replication lag and a WAL spike at 02:00

```
-- the nightly retention job
DELETE FROM events WHERE created_at < now() - interval '90 days';
```

`events` : 2,000,000,000 rows, 90 days retention, 6 indexes

daily rows removed	~22,000,000	one day's worth
dead tuples in the heap	22,000,000	DELETE only marks them
WAL at delete time	22,000,000	also replayed on every replica
transaction	hours	holding an xid and a snapshot the whole time (Mistake #2)
vacuum		WAL traffic (heap + all indexes)

Detection

```sql
-- which statement wrote last night's WAL spike?
SELECT queryid, calls, rows, wal_bytes, total_exec_time
  FROM pg_stat_statements
ORDER BY wal_bytes DESC
LIMIT 5;

-- dead rows by table
SELECT relname, n_dead_tup, n_live_tup, last_autovacuum
  FROM pg_stat_user_tables
ORDER BY n_dead_tup DESC
LIMIT 5;
```

- `wal_bytes` over 1 GB for one statement with `rows` in the millions
- `n_dead_tup` above a fifth of `n_live_tup` and `last_autovacuum` hours old (dead rows).
- Until you can partition: bounded batches, commit each one

The fix

```sql
-- events already exists. Build the partitioned table next to it; the key must include created_at
CREATE TABLE events_p (LIKE events INCLUDING DEFAULTS, PRIMARY KEY (id, created_at))
  PARTITION BY RANGE (created_at);
CREATE TABLE events_p_2026_08 PARTITION OF events_p FOR VALUES FROM ('2026-08-01') TO ('2026-09-01');
-- one partition per month back to the retention edge, indexes on events_p; pg_partman keeps the calendar
INSERT INTO events_p SELECT * FROM events WHERE created_at >= '2026-08-01' AND created_at < '2026-09-01';   -- a month per batch
SET lock_timeout = '2s';                                                    -- the swap is two renames
BEGIN; ALTER TABLE events RENAME TO events_old; ALTER TABLE events_p RENAME TO events; COMMIT;
-- from now on retention is a catalog change. PG 14+. not in a transaction block. waits for open transactions (Mistake #2)
ALTER TABLE events DETACH PARTITION events_p_2026_08 CONCURRENTLY;
DROP TABLE events_p_2026_08;
```

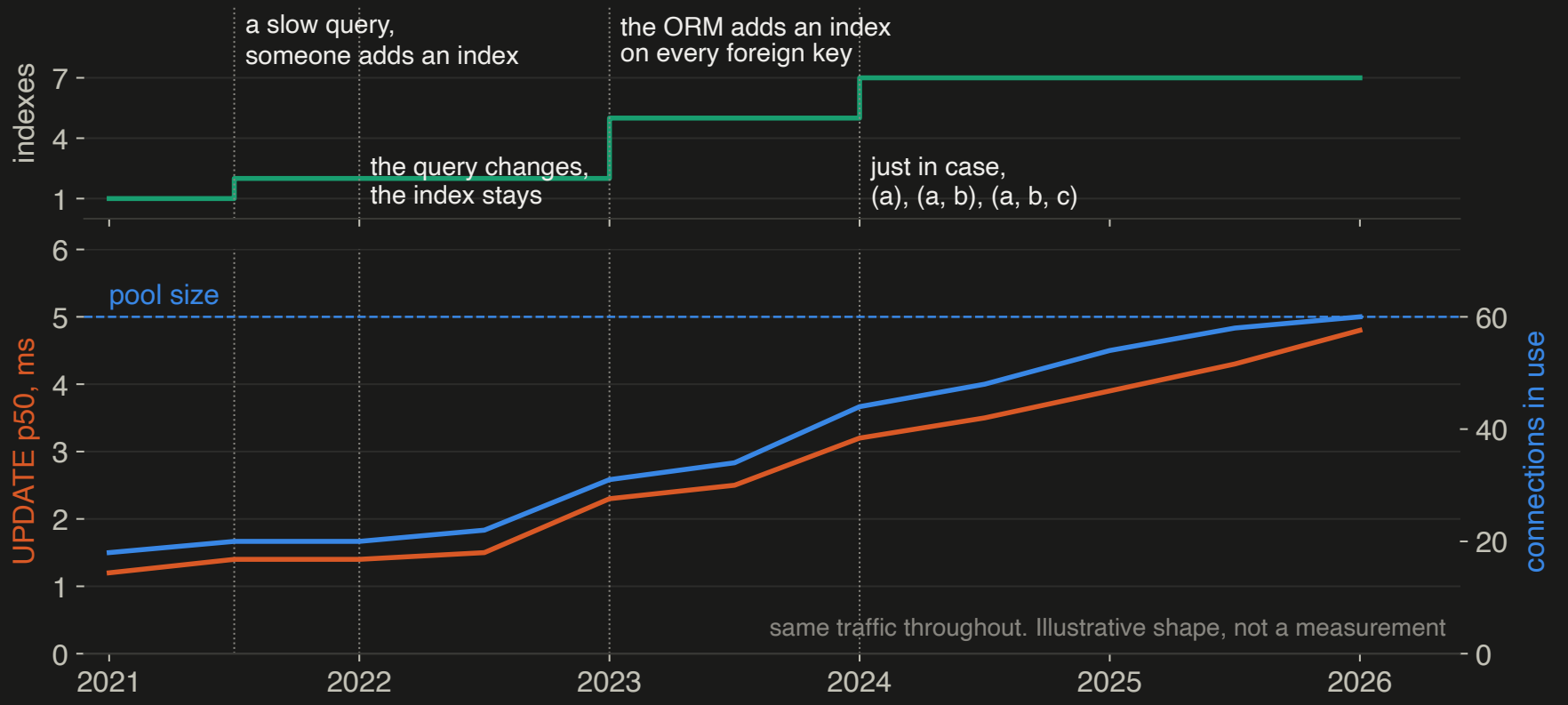
rows removed	~680,000,000, a month	
dead tuples	0	
WAL	catalog rows only	
wall clock	milliseconds, once open transactions on `events` finish	
alerting today	dead tuple ratio > 20% on large tables; replication lag	yes, hours later
	the statement itself (WAL bytes per statement, rows deleted)	no
verify	`n_tup_del` stays flat on the parent; the drop is a catalog change	

It's always connections

- The bulk `DELETE` is one long-running transaction. Its xid pins the horizon for hours
- Vacuum is held back the whole time. Dead versions pile up in every table, not just this one
- The index pages the `DELETE` empties stay allocated. Every `INSERT` and `UPDATE` walks a bigger index
- Slower writes hold their connection longer. Same traffic, longer holds, the pool opens more

Mistake #4: unused indexes

- UPDATE latency slowly increasing
- Connection counts up 1% month-over-month



Write amplification

Measured on 1M rows. Extrapolated to 1B

	1M rows, measured	1B rows at 5,000 UPDATE/s
heap + primary key on disk	179 MB	179 GB
seven int indexes on disk	141 MB	141 GB
page writes per UPDATE, primary key only	2	10,000 per second
page writes per UPDATE, seven indexes	9	45,000 per second
WAL per UPDATE, primary key only	311 B	134 GB per day
WAL per UPDATE, seven indexes	855 B	369 GB per day

- 1M: `UPDATE` takes 0.2 ms with 0 indexes or 7. Everything is in RAM
- 1B: indexes compete with heap for RAM.
- The `UPDATE` holds its connection through every one of those reads
- Every replica and the WAL archive carry the extra 235 GB a day

Detection

```sql
-- indexes with zero scans since the stats reset, largest first
SELECT s.relname, s.indexrelname, s.idx_scan, s.last_idx_scan,      -- last_idx_scan: PG 16+
       pg_relation_size(s.indexrelid) AS bytes
  FROM pg_stat_user_indexes s
  JOIN pg_index i USING (indexrelid)
WHERE s.idx_scan = 0
  AND NOT i.indisunique
  AND NOT i.indisprimary
ORDER BY pg_relation_size(s.indexrelid) DESC;
```

- `idx_scan` = 0 with `bytes` over 100 MB: a write tax on every INSERT and non-HOT UPDATE
- Zero counts only since `stats_reset` in `pg_stat_database`; a short window is not proof
- Each replica counts its own scans; check every node before dropping

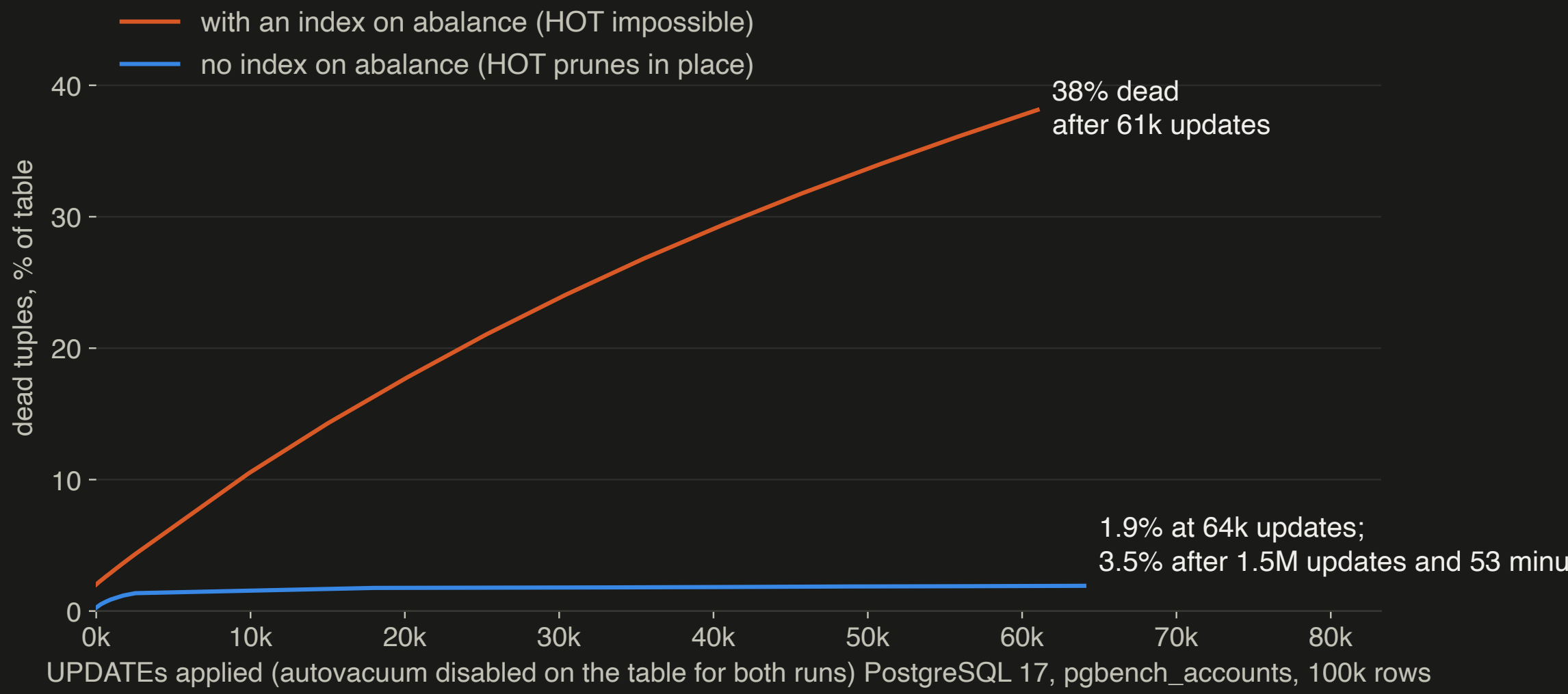
The index that turns off HOT

HOT: Only update the HEAP (Heap Only Tuples) if the new version fits on the same page and no indexed column change

```
UPDATE pgbench_accounts SET abalance = abalance + 1 WHERE aid = $1;
CREATE INDEX ON pgbench_accounts (abalance);                    -- second run only
```

	no index on `abalance`	one index on `abalance`
update	HOT update	every update changes `abalance` → no HOT
indexes	no index touched; old version pruned in place on the next visit	an index entry every time
dead tuples	stay flat, even with autovacuum off	every update leaves a dead tuple behind

One extra index: 38% dead vs 2% after 61k updates



Watch the HOT ratio, then drop the index

```sql
SELECT relname, n_tup_upd, n_tup_hot_upd, n_tup_newpage_upd,   -- n_tup_newpage_upd: PG 16+
       n_dead_tup
  FROM pg_stat_user_tables
WHERE n_tup_upd > 0
ORDER BY n_tup_upd DESC;
```

- `n_tup_hot_upd` under half of `n_tup_upd`: an indexed column is updated, or the page is full
- Counters are cumulative; diff two samples
- Save `pg_get_indexdef()` first, then `DROP INDEX CONCURRENTLY`

alerting today	unused index bytes (daily); invalid indexes (hourly)	yes
	HOT ratio	no
verify	`hot_pct` recovers on the next sample; the index is gone from `pg_index`	

It's always connections!

Actually - it's efficiency

Thirty years of the same response, raise `max_connections`.

- Just raising your connections glosses over the problem.
- Make every interaction with the database direct, fast and with consistent latency
- Your *app* depends on external services. Don't force your DB to as well.

Mistake	What it does to connections	The efficiency fix
connection storms	one per request, the CPU forks	Properly Sized pool
ignoring MVCC	every reader steps over the dead versions	a key that only grows gets its own table
long-running transactions	held for the whole wait	only SQL inside the transaction
bulk DELETE	one held for hours, every query slower	partition and `DROP`
unused indexes	every write holds longer, the pool grows	no index without a `SELECT`

Make each transaction cheaper and the connection count follows

Questions?

Scott Mead · scott@meads.us

Slides and the SQL from this talk will be posted: https://www.meetup.com/postgres-nyc/

Slides: Creative Commons Attribution License (CC BY 4.0). SQL and scripts: PostgreSQL License.

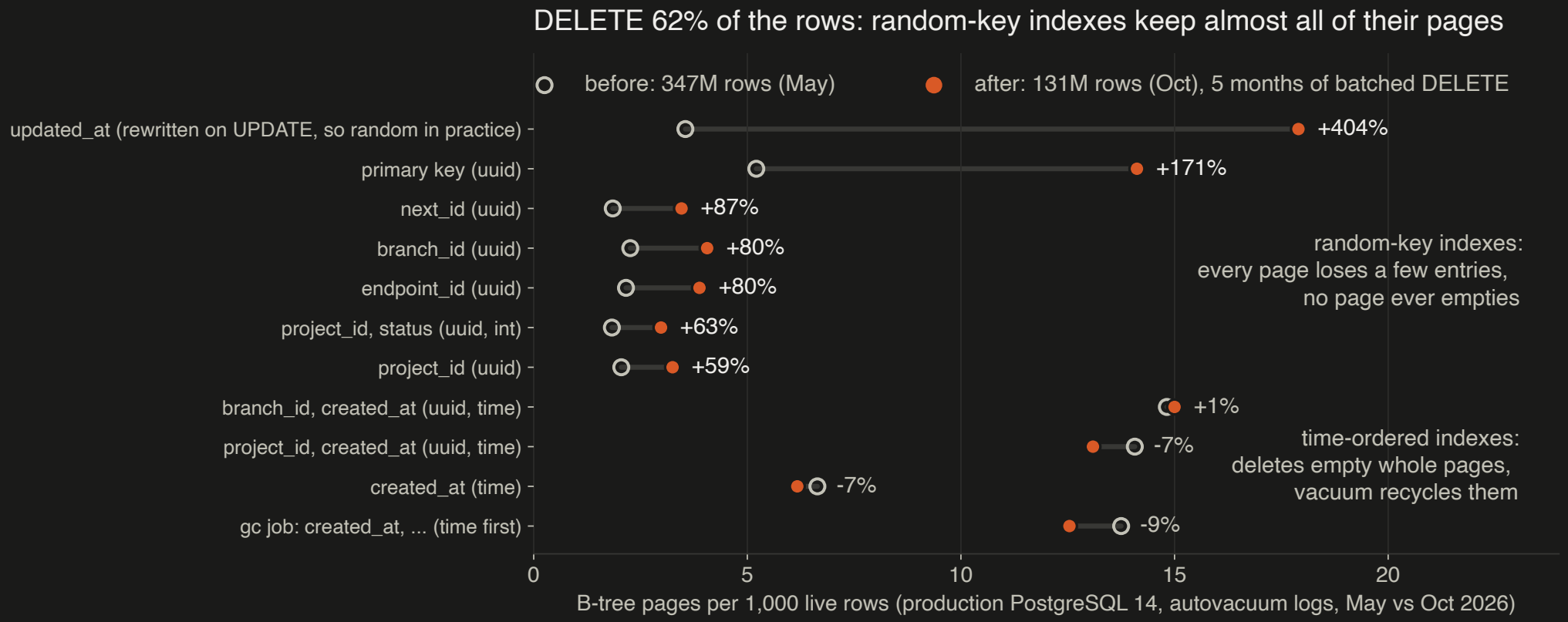
What vacuum tells you

```
INFO:  finished vacuuming "app.public.orders": index scans: 0
tuples: 0 removed, 1483112 remain, 612004 are dead but not yet removable
removable cutoff: 982144, which was 61930 XIDs old when operation ended
```

"dead but not yet removable" with an old cutoff: someone holds the horizon.

Holder	What it pins	Where to look
Report running 40 minutes *	snapshot (`xmin`)	`pg_stat_activity`
Nightly batch in one transaction	xid and snapshot	`pg_stat_activity`
`pg_dump` at noon	snapshot	`pg_stat_activity`
Physical slot with `hot_standby_feedback`	`xmin` for every table	`pg_replication_slots`
Logical slot nobody consumes	`catalog_xmin` ; WAL is retained	`pg_replication_slots`
Prepared transaction nobody finished	`xmin` , indefinitely	`pg_prepared_xacts`

* A read-only transaction holds one too.



Appendix: all eleven indexes behind the page diagram, May vs October 2026.

Appendix: know your defaults

You see nothing, until one query holds a connection for hours.

- `statement_timeout`
- `lock_timeout`
- `idle_in_transaction_session_timeout`
- `transaction_timeout` (PG 17)
- `idle_session_timeout` (PG 14)

All five default to 0: a connection can be held forever.

```sql
ALTER ROLE reporting  SET statement_timeout = '15min';
SELECT rolname                                          -- login roles with no statement_timeout override
  FROM pg_roles r
WHERE rolcanlogin
  AND NOT EXISTS (SELECT 1 FROM pg_db_role_setting s
                  WHERE s.setrole = r.oid AND s.setconfig::text LIKE '%statement_timeout%');
```

alerting today	login roles with unlimited timeouts	no
verify	the query above returns no application roles	