5 Wrong Hypotheses about PostgreSQL Multi-Transaction locks

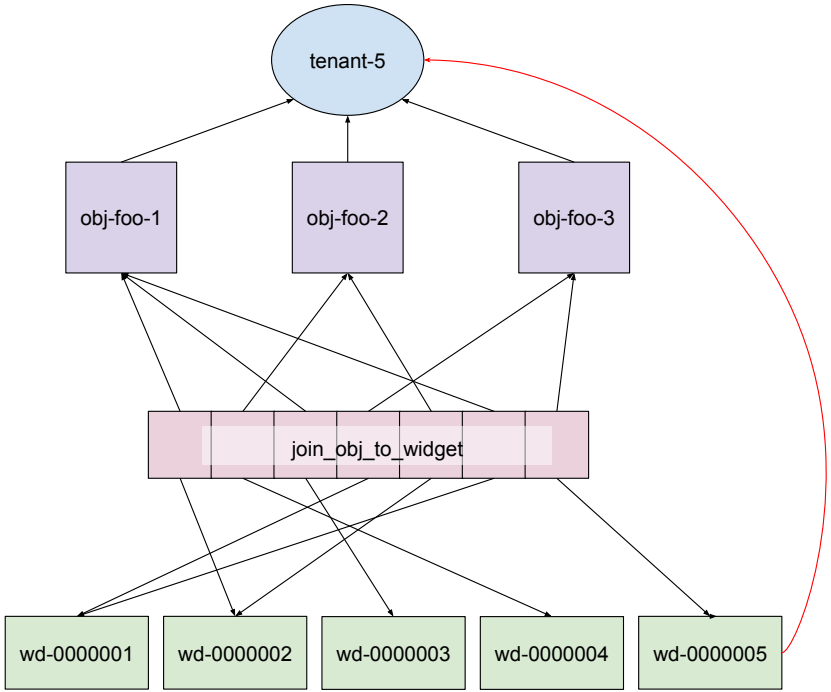
Who am I?

- Incident Nerd
- Tech Lead for a Production Engineering Team
- Podcast co-host for This is Fine! – Ask us questions about Resilience Engineering in Software at https://thisisfinepod.com!
- Just me, I do not speak for my employer

What are we going to talk about?

Site Reliability *SCIENCE* vs. Site Reliability ENGINEERING

PostgreSQL Multi-Transaction (MultiXact) locks

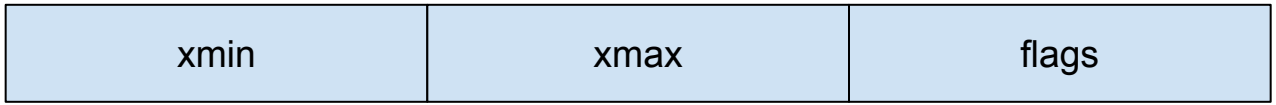


- Sub-transactions
- Foreign keys (see left)
  - inserts to widgets has to share lock tenant so it doesn't disappear during transactions

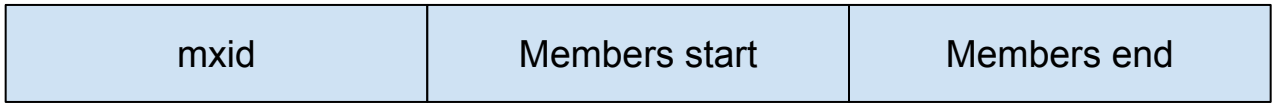
DB header fields

Table header fields

Tuple (row) header fields

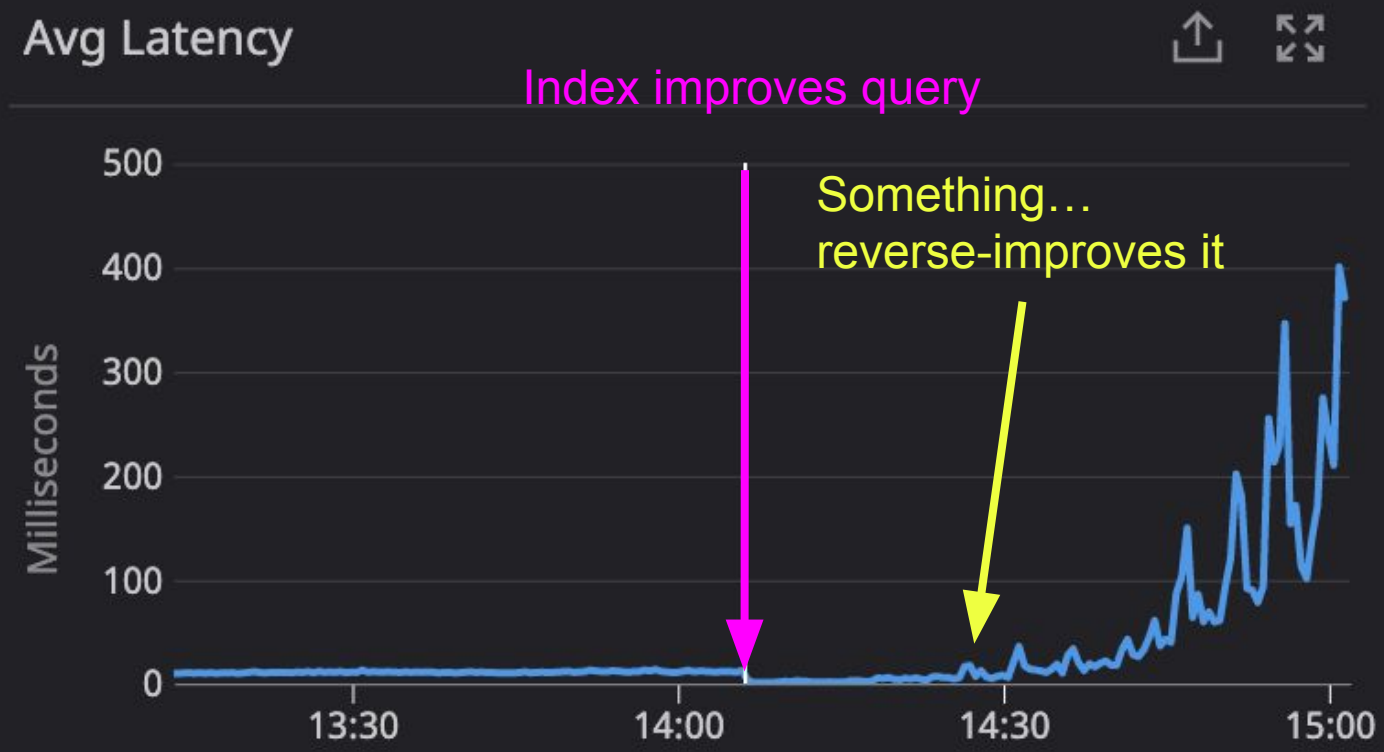


Offsets File Format

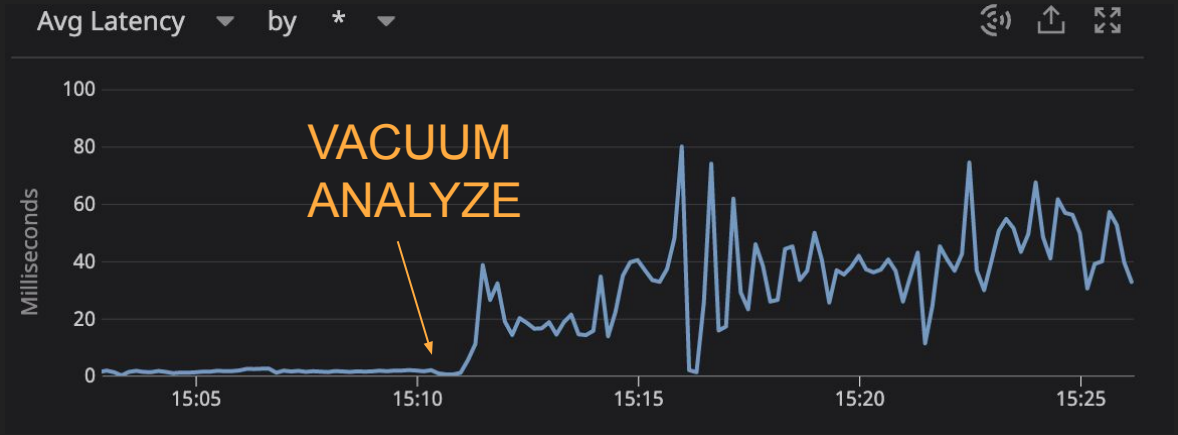


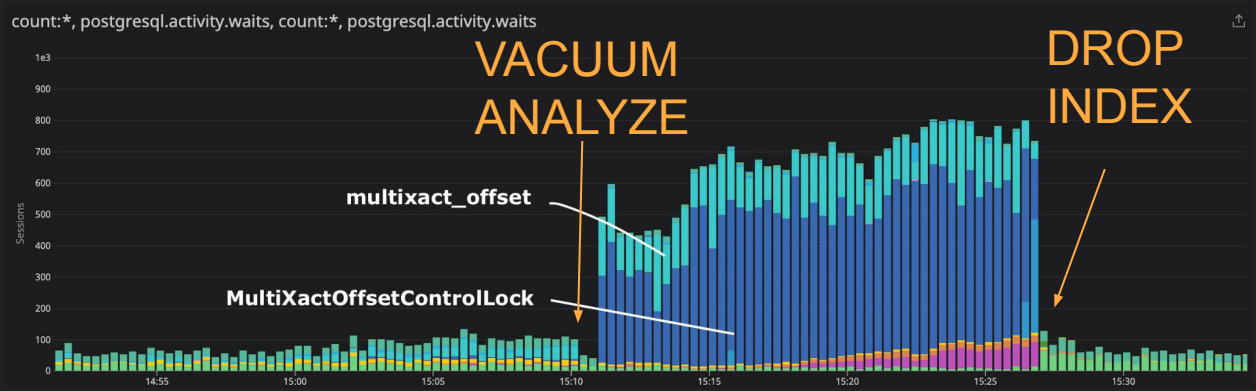
Members File Format

An index made a query faster, briefly



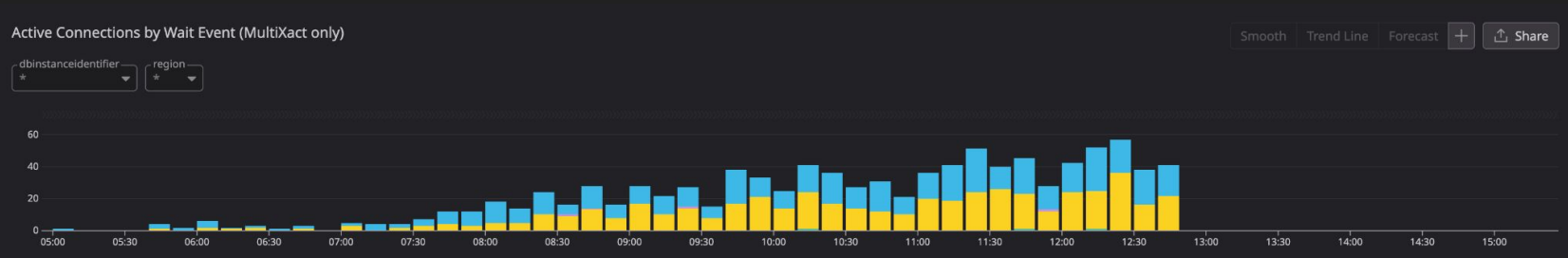
An index made a query faster, briefly, then made **everything** REALLY slow?





From judgement to learning

- “The root cause” was misleading
- SREs couldn’t explain LWLock:MultiXact* waiting threads
- Analysis uncovered hundreds of events like below, with automated recovery



Hypothesis 1

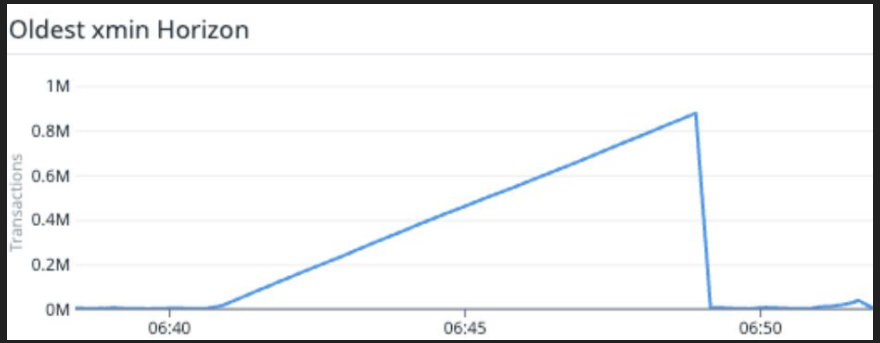
MultiXacts are bad.

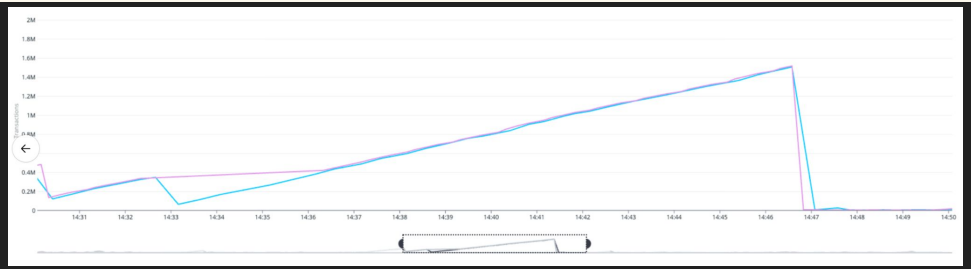
Experiment: Watch for MultiXact wait states correlated with failure modes.

Hypothesis 1 confirmed!

15:04 UTC `Oldest XMIN Horizon` starts to climb as part of a sustained trend.

15:16 UTC Posgres locks & wait events start to climb.





Hypothesis 2

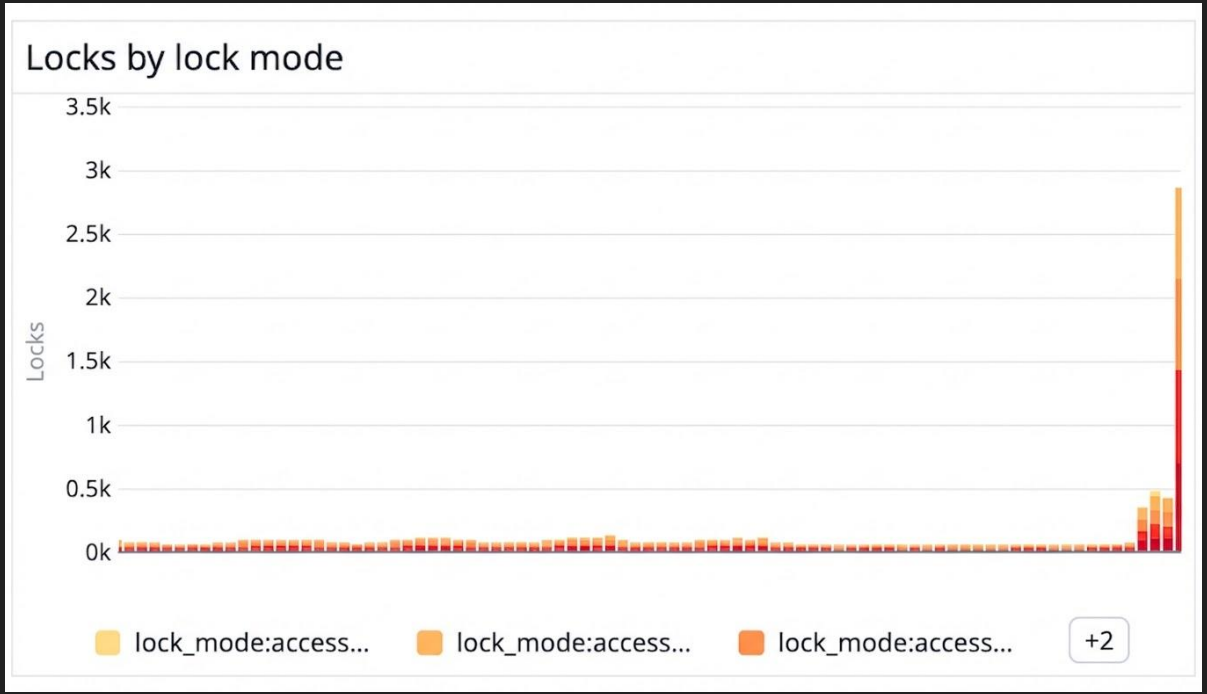
Long-running transactions are preventing autovacuum from cleaning out MultiXacts, causing them to build up in the system.

Experiment: Whenever we see these wait states, we can kill the longest running transactions and the system should self-recover.

Observations: Brief lockups over the next year, each one resolved by manually killing a few long-running queries.

Alert: Lock levels high!

- 2 years after that initial lockup, as traffic is ramping up, the app shows signs of trouble



To the runbook!


```
SELECT pg_terminate_backend(pid), query, pid, *
from pg_stat_activity where now() - xact_start >
interval '1 second' and query not like
'%autovacuum%';
```

Transactions stopped.. stopping

- pg_terminate_backend() wasn't working – queries kept holding their state: LWLock:MultiXactOffsetSLRU
- Team gives up and decides on the "nuclear option" – Stop the entire application for a few minutes and reboot everything

AI generated image w/ Gemini Pro

Everything goes green

AI generated image w/ Gemini Pro

Everything goes green… for 10 minutes

1000 hypotheses bloom

The database returns to full and total lockup state. Team generates hypotheses rapidly

- *Audit log inserts in the database are causing excessive transaction churn* - Insufficient observability to confirm.
- *Vacuums of high-churn tables eating all IO bandwidth* - Turned off but system must be healthy to test.
- *Low-priority background jobs are interfering* - Turned off but system must be healthy to test.

The team, out of options, turned as much as they could off, and went nuclear again.

Everything goes green

AI generated image w/ Gemini Pro

Everything goes green… for 24 hours

- **Over the next 8 days it happened 4 more times**
- Each time the response team got better at detecting and mitigating. A leading indicator was identified:

Generally ~ 5 minutes before total collapse: Table-level locks, and MultiXact*SLRU wait states would start increasing rapidly. Re-deploy fast enough and nothing breaks.

Hypothesis 2 invalidated

Long-running transactions are preventing autovacuum from cleaning out MultiXacts, causing them to build up in the system.

Result: Even w/ no long-running transactions, the system still locks up with MultiXact related wait states.

Hypothesis 3

The # of MultiXact locks being generated is somehow overwhelming the DB. If we generate fewer, the system will remain stable.

Experiment: Drop expensive foreign key constraints on high-churn high-concurrency tables, introduce concurrency limiters for transactions that share parent keys. Allow pausing low-priority work that might generate MultiXacts.

Expected result: Database does not fill up with locks anymore.

Expensive Experiment

Hypothesis 3 was born in fire and built on the idea that the # of MultiXact locks being generated was the problem.

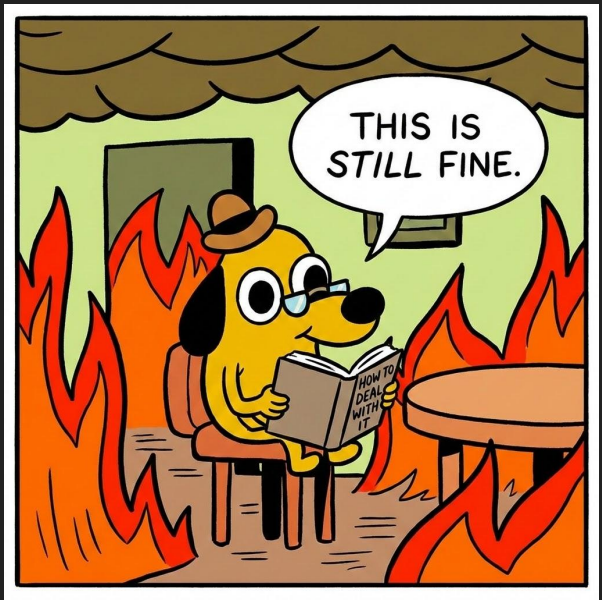
A fatal flaw: No measurement was known for the rate of MultiXact generation other than “the database won’t lock up anymore”. The only evidence our SRE scientists could observe was quiet pagers.

Hypothesis 3 Invalidated

The # of MultiXact locks being generated is a problem. Generate fewer and the system will remain stable.

Just a few weeks later, the database filled up with locks and MultiXact*SLRU wait states again. Low-priority work was paused to no avail, only re-deploying cleared the jam.

When in doubt: Read the literature!



AI generated image w/ Gemini Pro

- Busy SRE team had low-expertise on PostgreSQL and even less on Aurora PostgreSQL begins poring over blogs and docs
- Eureka! AWS had at some point published guidance on what to do about LWLock:MultiXact*SLRU wait states. It started with upgrading to PG16, which would allow observing and increasing the MultiXact buffer sizes – by a lot.

Hypothesis 4

Aurora PostgreSQL's handling of MultiXacts is different from community PostgreSQL: It has no OS level VFS layer to absorb MultiXact IO, which means cache-misses are slow and a high hit-rate on the cache is critical to preventing pileups.

Experiment: Upgrade to Aurora PostgreSQL 16 and increase MultiXact cache size by a factor of 16. Expect SLRU hit rate drop when there are continued lock pileups.

Hypothesis 4 is less wrong, but still wrong

Aurora PostgreSQL's handling of MultiXacts is special: no OS level VFS layer to absorb MultiXact IO means misses are slow, and a high hit-rate on SLRU caches is critical to preventing pileups interacting with them.

Experiment: Upgrade to Aurora PostgreSQL 16 and increase MultiXact buffers by a factor of 16. Observe SLRU hit rate drop when there are continued lock pileups.

Result: 0 lockups of this kind on the database from user activity. **However, hit rates plummeted below 60% a few times, with no MultiXact offset wait states.**

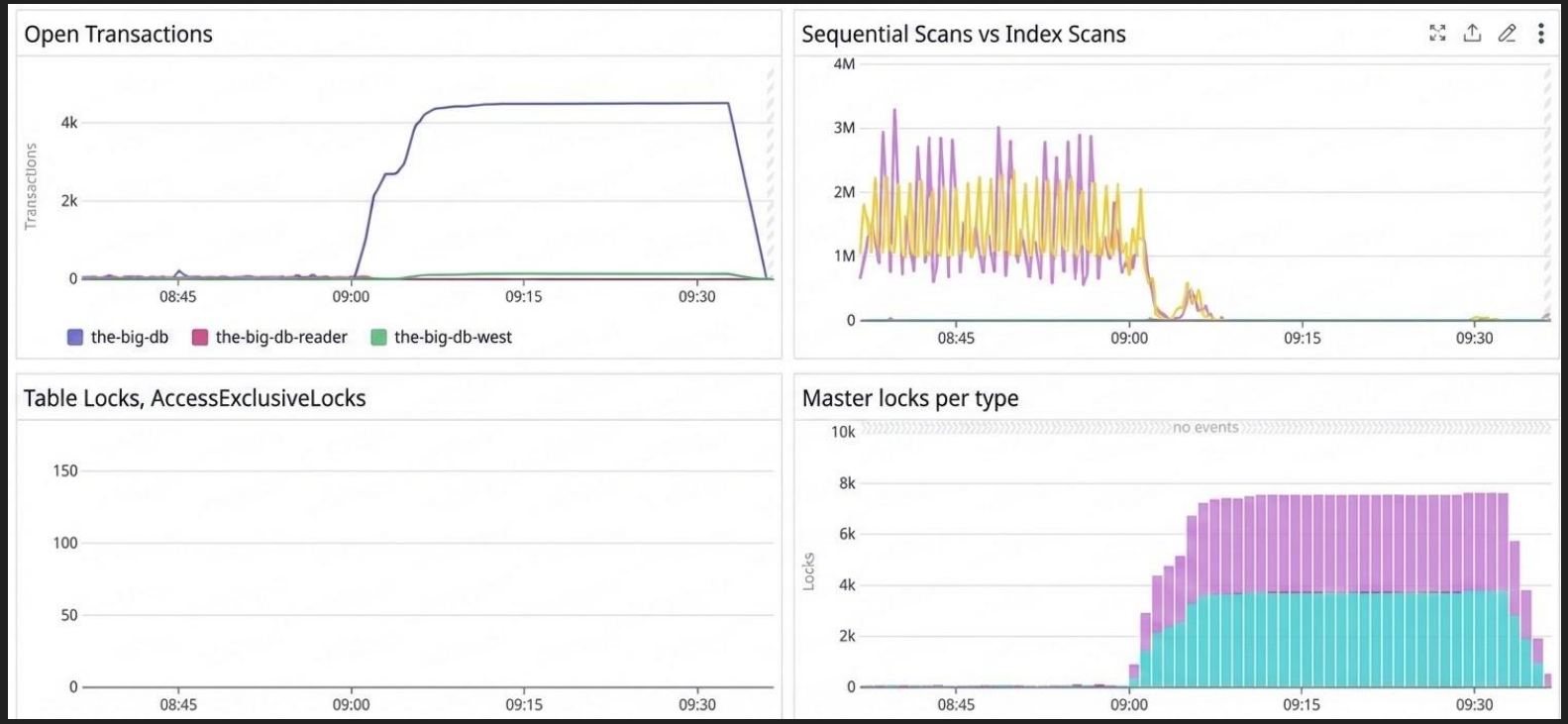
Spend your surplus error budget on science

- MORE BUDGET =~ MORE SCIENCE
- Modest load tests scheduled for production with a year of confidence in the hypothesis.
- TEST IN PROD OR LIVE A LIE

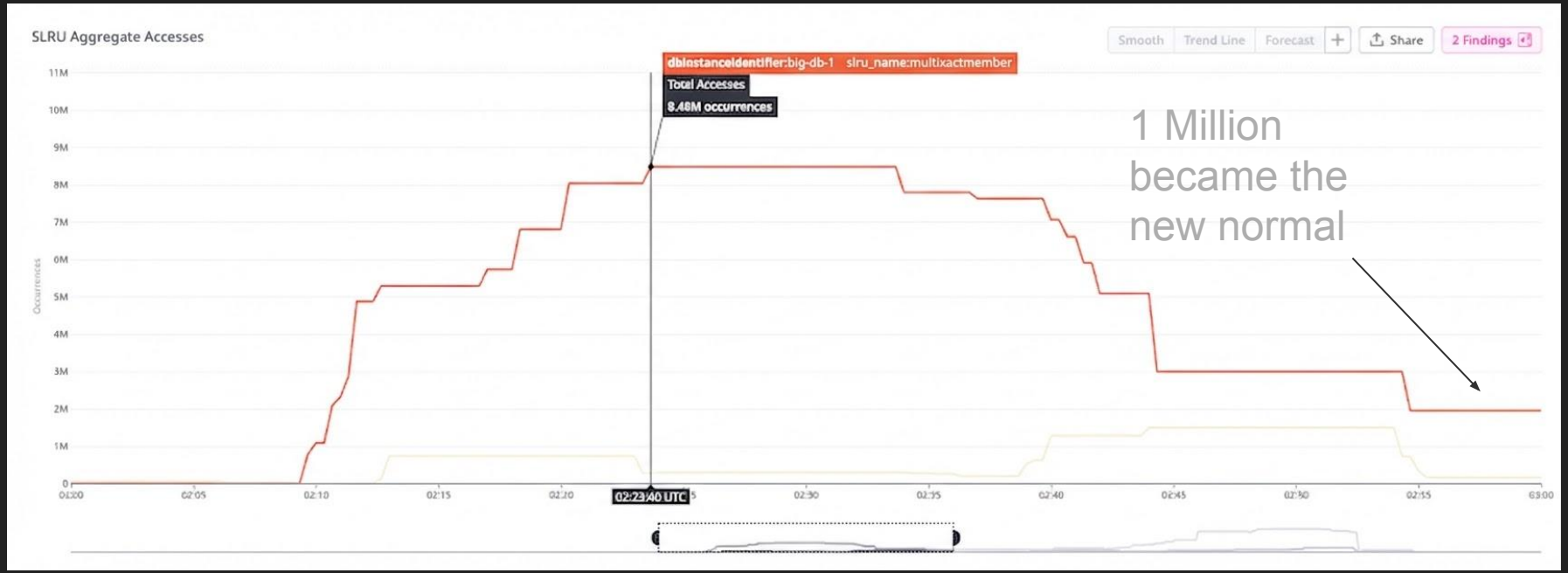
Traffic: +500%
SLRU hit rate: -2%

A few.. minutes.. later..

A few weeks later…

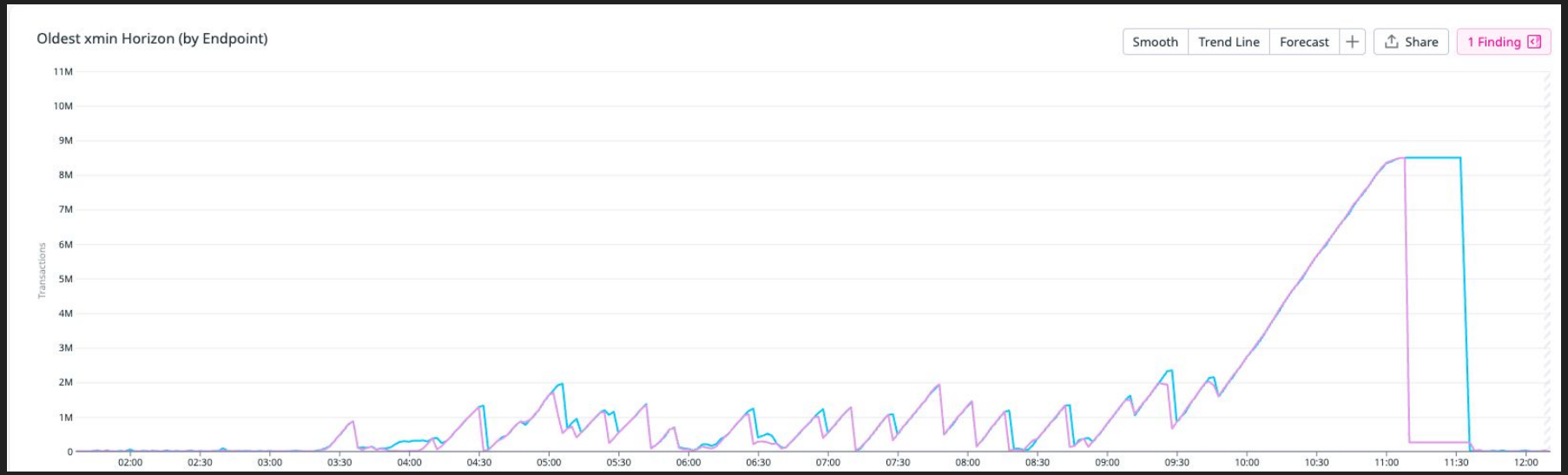


Residual pressure remained in the DB!



Triggered by an old pattern

A 13 minute transaction prevented vacuum from cleaning things up just long enough



Hypothesis 4 invalidated

Aurora PostgreSQL's handling of MultiXacts is special: no OS level VFS layer to absorb MultiXact IO means misses are slow, and a high hit-rate on SLRU caches is critical to preventing pileups interacting with them.

Experiment: Upgrade to Aurora PostgreSQL 16 and increase MultiXact buffers by a factor of 16. Observe SLRU hit rate drop when there are continued lock pileups.

Result: 0 lockups of this kind on the database from user activity **for a while**. However, hit rates plummeted below 50% a few times, with no MultiXact offset wait states. Meanwhile **during load tests hit rates remained above 98% and still lock pileups happened**. Residual effects seemed to remain present in the database, with the DB locking up again when aggregate SLRU accesses spiked.

Hypothesis 5

Aggregate MultiXact cache pressure can increase to the point where it overwhelms the system's ability to read rows with MultiXacts on them. VACUUM FREEZE relieves this pressure by freezing old MultiXacts removing the need to lookup their XID members.

Experiment: Alert on SLRU aggregate access levels and run VACUUM FREEZE if they exceed observed limit: 200k/s. Begin monitoring mxid age of all tables and observe effects of user activity and freezes on it.

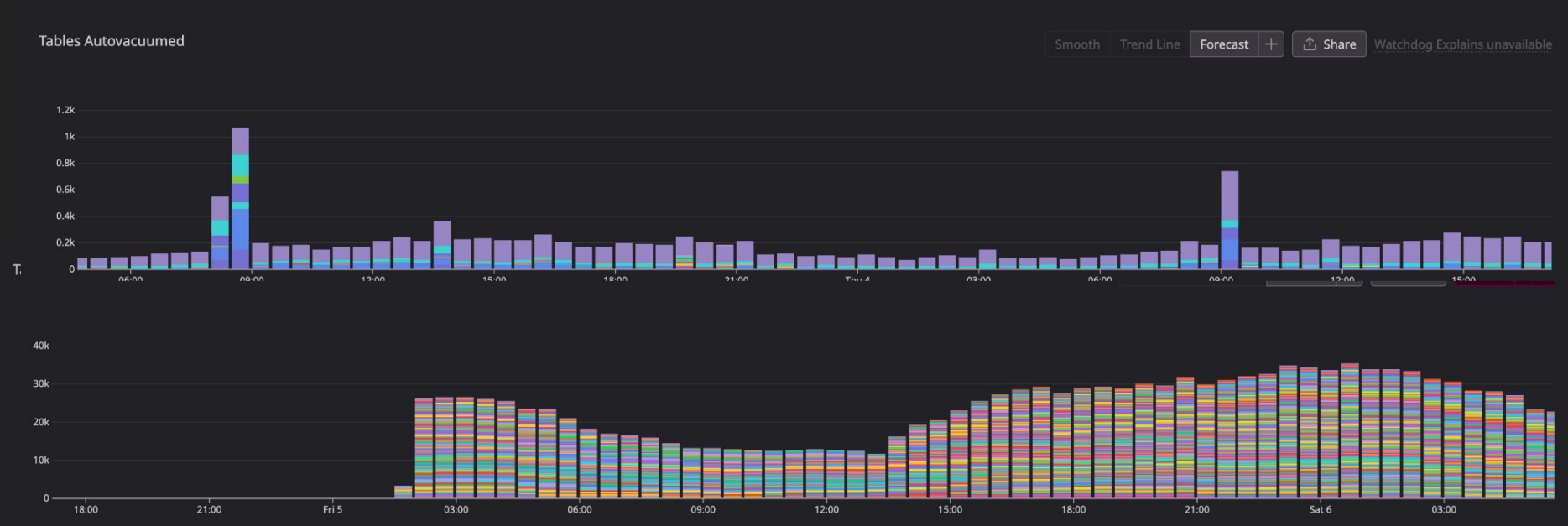
autovacuum_multixact_max_freeze_age has entered the chat

Throughout the journey, we wondered why autovacuuming wasn't helping more with this.

We learned that regular vacuums don't always freeze MultiXacts. But this threshold would trigger a VACUUM FREEZE if the mxid_age of a table's oldest unfrozen mxid (relminmxid) crossed a threshold.

- Default: 400,000,000
- We choose: 1,500,000
- Alerts on volume and age cease.

Holy autovacuum batman - Autovacuum 50X’s



Hypothesis 5 still kinda wrong

Aggregate MultiXact buffer pressure can increase to the point where it overwhelms the system's ability to read rows with MultiXacts on them. VACUUM FREEZE relieves this pressure by freezing old MultiXacts removing the need to lookup their XID members.

Experiment: Alert on SLRU aggregate access levels and run VACUUM FREEZE if they exceed observed limit: 200k/s. Begin monitoring mxid age of all tables and observe effects of user activity and freezes on it.

Result: System stays up. However, for reasons, it has exceeded 2 million and no lockup. Where is the line? WE NEED IT TO BREAK

Hypothesis 6?

Why does a manual VACUUM FREEZE on tables that have been auto vacuumed multiple times drop the age?

What exactly are we doing to make so many MultiXacts?

How much are pg17’s banked MultiXact buffers improving our robustness to this problem

Are we wasting IO on over-vacuuming?

Detailed Incident Reports make better hypotheses

Know your error budget and run realistic experiments when you have a surplus!

Don’t be afraid to be a little wrong, just try to get LESS wrong