Episode Book

EPISODE #6622

How Far Postgres Actually Scales in Production

PUBLISHED	16 November 2026 at 00:36 UTC
DURATION	26:35
LISTEN	myweirdprompts.com/episode/postgres-scaling-limits-production-01006/
HOSTS	Corn, Herman, Dorothy, Hilbert
SCRIPT MODEL	z-ai/glm-5.3
VOICES	omnivoice
PIPELINE	V5.33
BOOK GENERATED	16 November 2026 at 02:53 UTC

SYNOPSIS

ChatGPT's backend serves 800 million users on a single unsharded Postgres primary, proving that database ceilings are rarely about raw capacity. Instead, production teams hit operational limits like vacuum, connection counts, and lock queues. This episode examines how OpenAI keeps one writer alive with nearly fifty read replicas and five-second schema timeouts, alongside why Instagram and Notion chose logical sharding—using Postgres schemas, self-describing IDs, and divisible shard counts—to tame CPU spikes and transaction ID wraparound.

TRANSCRIPT



Corn

Eight hundred million users on a database nobody sharded. That's my kind of reckless.



Corn

Daniel's prompt this time is one of those quiet grenades. He wants to know how far Postgres actually scales, and he's guessed the answer is further than any startup could ever imagine, but he doesn't want benchmarks or theory. He wants real stories, documented production deployments, and specifically the moments where the people running them found out where the real limits were. Which is the right way to ask it, because the answer isn't a number.



Herman

It never is with databases, and that's what I love about this question. The most extreme documented case is also the most surprising one, so let's start there.



Corn

Go.



Herman

ChatGPT's backend. Eight hundred million users, and the relational core of it is a single unsharded Postgres. One primary on Azure Database for PostgreSQL, taking every write, with nearly fifty read replicas spread across regions. Load on that system grew more than tenfold in a single year, and reads held a p99 latency in the low tens of milliseconds the whole time.



Corn

One primary. Every write for eight hundred million people, through one door.



Herman

One door, and they left it that way on purpose. That's the detail I want to sit on, because everything else in this episode is a story about people who did shard, and OpenAI is the story about a team that looked at the sharding question and said, no, we'll scale the reads instead. Reads go to replicas and a cache. Writes stay on the primary.



Corn

Which is the whole trick, and also the whole weakness. One writer is one point of failure.



Herman

And they say so themselves, which I respect. Their own engineering write-up is very direct about where the limit lives. New workloads that are write-heavy and shardable don't go to Postgres at all, they go to sharded systems like Cosmos DB. Postgres keeps the relational data that needs strong consistency. And that's the moment they found the edge. Not a crash. A routing decision.



Corn

They hit the wall of one writer and simply chose not to climb it.



Herman

Which, to be fair to the wall, is the correct engineering response. But before anyone thinks this is easy, the guardrails they need around that single primary are a whole discipline on their own. PgBouncer in transaction mode so connections get multiplexed instead of each app worker holding its own connection. Workload isolation, so a heavy analytics query can't starve the production path. Rate-limited backfills. Timeouts on idle connections and long-running transactions.



Corn

Don't let anything sit there holding a transaction open.



Herman

Because a long-running transaction blocks vacuum from cleaning up row versions, and we'll get to why that's the thing that actually kills you. They even run a five-second timeout on schema changes, so a migration can't lock a table out from under eight hundred million users.



Corn

Five seconds. My last schema change took three meetings and an apology email.



Herman

And it's worth spelling out why that timeout is so aggressive, because it's not paranoia. A schema change that needs an exclusive lock has to queue behind every running query on that table. And while it waits, everything behind it queues too. So one slow query plus one migration equals what looks like a total outage from the application's point of view, even though the database is technically fine. The five-second cap means the migration just fails and retries instead, and a failed migration is a Tuesday, but a lock pile-up at that scale is a headline.



Corn

So the timeout isn't protecting the schema change. It's protecting the queue behind the schema change.



Herman

Exactly right. The failure mode was never the migration itself, it's everything the migration accidentally blocks. They're also testing cascading replication now, where replicas relay the write-ahead log onward to other replicas, so the primary feeds fewer of them directly. That's a fan-out problem. If one primary has to feed fifty replicas directly, that's a lot of network bandwidth coming off the box that's also doing all your writes. Cascade it and the primary feeds a handful, and those feed the rest.



Corn

The tree doing the lifting so the trunk doesn't snap.



Herman

That's nicer than the metaphor I'd have used, so I'll take it. And the incident record is the part that quietly amazes me. The talk their engineer gave at PGConf reported one top-severity Postgres incident in nine months, running at that scale. One. Nine months.



Corn

One bad night in nine months. I've had worse months doing nothing.



Herman

Now, one honesty flag before we go further, because this figure is everywhere and it shouldn't be. Secondary write-ups about that deployment say millions of queries per second. That's a headline. I couldn't find an exact number behind it anywhere, and I wouldn't repeat it as a fact. What's documented and defensible is the unsharded primary, the roughly fifty replicas, the tenfold growth, and the p99 numbers. That's already the most extreme single-primary deployment anyone's published. It doesn't need the inflated figure.



Corn

The real story is so absurd the exaggeration is redundant.



Herman

The load grew ten times in a year and they kept one primary. Let that be the headline.



Corn

Herman, can I ask the skeptic's question, since you'll enjoy it. Would you trust that shape? One primary, everything consistent through it, fifty replicas holding the reads.



Herman

The clinician's answer is no, and then the engineer's answer is yes with monitoring. My instinct, from years of medicine, is deep distrust of any system that reports itself fine. A single primary doesn't report itself fine. It reports exactly what it is, which is one box



Herman (continued)

whose failure is everyone's failure. So you make that box boring. You guard it, you keep its workload predictable, and you accept that you've traded distributed risk for concentrated, watchable risk. OpenAI decided watchable was better. I think for their read-heavy workload they're right.



Corn

Concentrated but boring. That's the selling slogan of my entire life.



Herman

Which is why you'd defend it. I know you, you'd have kept that primary until the heat death of the universe and called it reliability.



Corn

I'd have kept it until someone proved the replacement could nap.



Herman

The one serious criticism of the shape, and it's structural, not operational. Scale reads all you like, the writes still queue through one door. The moment your workload tips from read-heavy to write-heavy, that architecture has nothing left to give you. And that's the pivot to the rest of the episode, because everything we're about to cover is what teams did when the write side, or a single table, outgrew that shape.



Corn

Before we get into the shards, let's hear from your mother.



Herman

Oh, yes, please, Mum's up today.

D

Dorothy

Hello dear, it's your mother, with Dorothy's History of Toast.

D

Dorothy

Melba toast is named after the opera singer Nellie Melba, and the chef Auguste Escoffier is credited with it, in the eighteen nineties.

D

Dorothy

A whole famous toast named after somebody, and she never asked for it, it just seems sad somehow, all that thin dry bread standing in for a person.

D

Dorothy

Anyway, sweetheart, I've got a loaf going stale on the counter, so I'll let you get back to it. Bye now.



Corn

A person reduced to dry bread. There's a database metaphor in there.



Herman

There is not, and we're not making it. So. The sharded half of the story. Start with the oldest famous one, Instagram, twenty twelve.



Corn

Before you do, one framing thing. The interesting thing in every one of these stories is never can Postgres hold a lot of data. It's that the ceilings people hit are operational. Vacuum. Transaction ID wraparound. Connection counts. Per-table size. The capacity of the engine is never the thing that bites first.



Herman

And that's worth saying up front because it reframes every story we're about to tell. Nobody runs out of Postgres. They run out of vacuum, or connections, or comfort. Instagram's version is the classic. Several thousand logical shards, and here's the part people miss, logical shards, meaning Postgres schemas, on far fewer physical servers. Exact counts are secondhand, their own post doesn't pin the numbers, but the shape is documented. Shard as a schema, not as a machine.



Corn

So the shard count is a fiction you control, not hardware you buy.



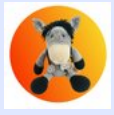
Herman

And that fiction is the whole point. Because the number of logical shards hugely outnumbers the machines, you can move shards between physical boxes later without re-splitting data. You're buying yourself future flexibility with a naming convention.



Corn

Hold on, let me make sure I've got that, because it sounds like accounting rather than engineering. If I've got, say, a thousand schemas spread over ten machines, and one machine gets hot, what actually happens?



Herman

You move schemas, not data boundaries. Because each schema is self-contained, you can detach a handful of them from the overloaded box and attach them to a fresh one, and no shard ever has to be split in half. If you'd sharded by hashing directly onto ten machines, growing to eleven would mean rehashing nearly everything. The extra zeros in the logical count are slack. It's the same reason a moving company sells you more boxes than you need. You never repack the whole truck to fix one bad box. The other famous detail from that post is how they generated IDs inside Postgres itself. Each identifier packs forty-one bits of timestamp, thirteen bits of shard, ten bits of sequence, so every ID carries the time it was made, which shard made it, and its order within that shard. No coordination needed between shards, because the ID is self-describing.



Corn

The primary key that tells you where it lives. No lookup table.



Herman

The database equivalent of an address with the postcode baked in. Notion is the modern version of the same instinct, and their story is better because their write-up is brutally honest about why they shattered. It wasn't capacity. Notion wasn't out of room. The triggers were vacuum stalling, the fear of transaction ID wraparound, and CPU spikes that were paging engineers. Their database was keeping them up at night.



Corn

They didn't shard for size. They sharded for sleep.



Herman

That's the honest summary, yes. So they split into four hundred eighty logical shards, again Postgres schemas, on thirty-two physical databases, fifteen shards per host. And the reason for four hundred eighty is my favorite detail in the entire sharding literature. They picked it because four hundred eighty has many divisors.



Corn

They chose the shard count by its factors.



Herman

Because if you ever need to redistribute, you want a number that divides cleanly into lots of groupings. It's arithmetic as future-proofing. Four hundred eighty divides by two, three, four, five, six, eight, ten, twelve, fifteen, sixteen, twenty, and more. So whether the future held thirty-two machines or sixty-four or a hundred twenty, the shards would spread evenly with nothing left over. They also set design caps at the outset. Five hundred gigabytes per table, ten terabytes per physical database. Explicit ceilings, written down before they were reached, which is the exact opposite of how most teams meet a ceiling.



Corn

Most teams meet a ceiling the way you meet a door in the dark.



Herman

Then twenty twenty-three, the Great Re-Shard. They went from thirty-two machines to ninety-six, no downtime. Users saw, at worst, about a second of spinner. Before the re-shard some shards had passed ninety percent CPU at peak and were near their IOPS limits. After, peak CPU was around twenty percent. And here's the bit that makes the whole episode's argument in one move. The new bottleneck after the re-shard wasn't the database. It was PgBouncer connection limits. They fixed it by splitting the poolers into four groups.



Corn

Fix one ceiling, the next one walks in the room.



Herman

And that's not a Notion failing, that's the pattern everywhere. Figma's version of the story is the vertical-then-horizontal one. Their database stack grew almost a hundred times since twenty twenty. It started as literally one Postgres on AWS's largest instance, which is such a startup move, just buy the biggest box and keep going. Then replicas, then caching, then by end of twenty twenty-two, about a dozen vertically partitioned databases, splitting by domain rather than by key.



Corn

Which buys years, and feels free at the time.



Herman

It feels free right up until some tables hold several terabytes and billions of rows, and then the limits show up, and again it's not raw capacity. It's vacuum, and the maximum IOPS their RDS instances could deliver. Their first horizontally sharded table shipped in September twenty twenty-three, after about nine months of work, and they built their own proxy for it. An in-house Go proxy called DBProxy over RDS Postgres. They actually evaluated the alternatives, the CockroachDB and Vitess style options, and went with their own.



Corn

Nine months. The same gestation period as a human, for one table.



Herman

A very large table, to be fair to them. And I find that choice telling, because it's the second time today the story is teams staying with Postgres and engineering around it rather than leaving. We said the same thing when we did the episode on Postgres absorbing the vector databases, killing the sprawl, and the pattern holds at this scale too. Nobody wants a second database. They want their database to keep working.



Corn

Because the second database isn't a database, it's a career.



Herman

Microsoft's entry is the biggest one by raw numbers, and it's Postgres wearing an extension. VeniceDB, their Windows diagnostics pipeline, built on Citus, which is the sharding extension for Postgres. Data from one point two billion devices, over twenty thousand distinct metrics, fifty-four nodes in two clusters, three and a half thousand cores, twenty-seven terabytes of RAM, and one point six petabytes of SSD. Running more than ten million queries a day, three quarters of them under a hundred fifty milliseconds.



Corn

One point six petabytes. And still the same engine under the hood.



Herman

Same engine, sharded by the extension rather than the application. Which is the interesting variation, because Instagram and Notion put the sharding logic in the app layer, and Citus puts it inside the database so the application mostly doesn't know. The application issues what looks like an ordinary query, and the extension decides which node holds which rows. You trade some control for not rewriting your app, and for a diagnostics pipeline ingesting from a billion devices, that trade makes sense.



Corn

So the app stays innocent and the database does the paperwork.



Herman

Microsoft bought Citus Data in twenty nineteen, which tells you how that went for them. And the pipeline that makes all of these work, PgBouncer, came out of Skype, back in two thousand seven. They open-sourced it along with PL/Proxy because they needed to multiplex connections across partitions. Nearly universal at scale now, and it started as one telecom's internal plumbing problem.



Corn

Every piece of infrastructure starts as somebody's Tuesday.



Herman

Now the counter-story, because this can't be an advert. Uber, twenty sixteen, the famous post about why they switched from Postgres to MySQL. Write amplification, where an update touches every index on the table, so an update isn't one write, it's one write per index. Heavy physical WAL replication between data centres. A replica corruption bug. Painful major version upgrades. Process-per-connection limits. All real problems.



Corn

The but is coming. I can hear it in your breathing.



Herman

The but is that all of it was on Postgres nine point two. And the rebuttals, including from EDB, pointed out that later versions and logical replication addressed a large part of what they'd hit. Logical replication in particular changed the cross-data-centre story entirely, because instead of shipping the physical page images, you ship the logical changes, and a table with ten indexes doesn't pay for all ten on every update across the wire. The lesson from Uber isn't Postgres can't scale. It's that the version and the replication model matter enormously, and a twenty sixteen complaint about a twenty twelve-era database isn't evidence about the one running ChatGPT today.



Corn

So the honest reading of the whole record is two shapes. One big primary plus replicas, taken absurdly far, or shard it yourself, logical shards as schemas, way more shards than machines. And almost nobody in the documented record leaves.



Herman

Instagram, Notion, Figma, Microsoft, all sharded Postgres itself. Uber is the exception that proves the version mattered more than the engine.



Corn

But the ceilings that forced every one of those decisions were never total data volume.



Herman

They were per-table size, vacuum, wraparound, connections. Paxos is the neatest recent example. They had a twenty-one terabyte ledger table on Aurora Postgres that had reached about two thirds of Aurora's per-table limit. And they partitioned it without moving a single row, by attaching the existing table as the historical partition of a new partitioned table. Cutover took three hundred seventy-one milliseconds. The first attempt failed, and the reason it failed is my favorite sentence in that whole write-up. A constraint validation was blocking autovacuum.



Corn

They tried to fix a table that was too big by running a check that made vacuum stop working.



Herman

Fix one ceiling, meet the next one. Their first attempt died on the very problem the partitioning was meant to relieve. And the detail I love about the successful attempt is how undramatic it was. Three hundred seventy-one milliseconds is less time than it takes to say the words "zero downtime migration." The whole thing is a transaction that swaps one table for two, and the application never knew. Tiger Data, formerly Timescale, claims close to three petabytes on their own Postgres-based monitoring service, over three trillion metrics a day, and I'd flag that as a vendor's own claim about their own extension, but it's a data point on where the engine sits. The ceiling is operational, and it moves every time you fix it.



Corn

You keep saying wraparound like it's a monster under the bed. Tell the actual stories, because that's the one that takes companies down.



Herman

It is. Every transaction in Postgres gets an ID from a counter of about two billion values, and old rows have to be frozen by vacuum before the counter laps them. If vacuum falls far enough behind, Postgres refuses new writes rather than risk confusing old data with new. That's a safety feature, and it looks exactly like an outage. Sentry, July twenty fifteen. Their autovacuum was poorly tuned, one run failed silently, and on a Monday the database stopped taking writes. They were down for most of a US working day. They got out by going read-only on replicas, failing over to bigger hardware and truncating one large table, and afterwards they doubled their autovacuum workers.



Corn

And the second one is worse, I'm guessing, because you've gone quiet.



Herman

Mailchimp's Mandrill, February twenty nineteen. One shard hit wraparound and went into safety shutdown. About forty hours before they could announce it was resolved. A full vacuum was estimated at up to forty days, so that was never an option. And the line in their post-mortem that stays with me: they'd spotted the risk in November twenty eighteen, and filed a ticket.



Corn

Same failure, four years apart, and the second team had read about the first.



Hilbert

Herman, the Mandrill one. The forty days. Did they ever say what the tables were, or just the size?



Herman

Terabyte-scale tables, two of them, and they truncated both rather than wait. The full vacuum was estimated at up to forty days, so truncation was the only sane move left by then.



Hilbert

Forty days for a vacuum. I had a run-in with that myself, the ledger.



Corn

The ledger?



Hilbert

The neighbour two doors down, Rosen. Sells shelving. Brackets, rails, screws, the whole wall of it. In twenty nineteen he asked me to keep his stock and invoices straight, so I put up a Postgres on a small managed instance, and I called it the ledger, because that's what it was. Bill went in my name. Rosen paid the first year and then developed amnesia about it.



Corn

And you kept paying.



Hilbert

Six years. It was a good instance. Boring. Ran itself, mostly. Anyway, the wraparound warning came up in the dashboard around the end of the third year, the age counter going orange, and I screenshotted it and set it as my desktop background so I'd have to look at it every day and deal with it. Which I did. Look at it, I mean. Dealing with it came later, after it went read-only for a week and Rosen rang me up asking why the till was guessing. Cost me a month of the bigger instance and a Saturday.



Herman

The screen background as a ticketing system. That's the most aggressive form of filing a ticket I've ever heard, and I've debugged a connection pool at three in the morning.



Hilbert

The ticket was filed, Herman. Filing isn't the same as doing, which is what you two were circling around all episode without saying. Mandrill spotted the wraparound risk in November and filed a ticket too. And their outage ran forty hours. Mine ran a week because I'm one man with a desktop wallpaper, theirs ran forty hours with a whole engineering department.



Corn

What happened to the ledger, in the end?



Hilbert

Rosen retired last spring. Didn't ask for the data. I shut the instance down in June and stopped paying for it, which was the first time in six years that bill did anything I wanted. The wallpaper's still there, mind. I've gotten used to the orange.



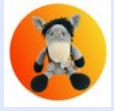
Herman

A monument to the ticket never closed.



Hilbert

It's a good orange. Anyway, you were saying about the ceilings moving.



Herman

Right, the common misconception, then, because it's sitting right here. People believe the ceiling is raw data volume, that Postgres dies when the petabytes arrive. The documented record says otherwise. Notion hit vacuum and wraparound fear, not capacity. Figma hit vacuum and IOPS limits with tables in the terabytes, while their total data kept growing happily. Paxos hit a per-table limit at two-thirds full. The capacity was always there. The



Herman (continued)

operations were what broke.



Corn

And the second half of the misconception, that teams at scale eventually leave. The record is the opposite. Instagram, Notion, Figma, Microsoft, all of them sharded Postgres itself rather than migrate off it. Uber's the famous departure, and it was version nine point two complaining.



Herman

So if you take one story from today, take the shape of the pattern. OpenAI kept one primary and scaled the reads. Everyone else stayed on the engine and split themselves. And the ceiling that moved each of them was never the one they expected.



Corn

Which leaves the question I'd actually put to any team listening. Not will Postgres scale, because the evidence says further than you'll ever push it. The question is whether you'll notice the ceiling before it notices you. Mandrill saw the warning in November and filed a ticket. Hilbert saw his every morning in orange.



Herman

And the direction of travel is clear. OpenAI is testing cascading replication and routing the write-heavy work elsewhere while the relational core stays. Notion re-sharded with a second of spinner. Figma built its own proxy. Everyone is engineering around the ceiling, not out from under it.



Corn

And a very big thank you, as ever, to our producer, Hilbert Flumingtop, for keeping the whole show running while we talk about databases that run themselves.



Herman

If you want more of this, try episode twelve twelve, The Postgres Vector Revolution; episode seventeen, Cloud Render Superpowers; and episode twelve thirty-three, Why Just Use Postgres Isn't Always Enough. This has been My Weird Prompts.



Corn

If you've enjoyed this one, the whole archive lives at my weird prompts dot com, all six thousand and change.



Herman

We'll be back soon. See you then.