

OpenAI: Scaling PostgreSQL to the Next Level

 tr_cn (/profile.php?user=tr_cn) • 2025-05-19 22:29:18 • 22,578 views 1 comment

At the PGConf.dev 2025 (<https://2025.pgconf.dev/>) Global Developer Conference, Bohan Zhang (<https://bohanzhang.me/#talks>) from OpenAI shared OpenAI's best practices with PostgreSQL, offering a glimpse into the database usage of one of the most prominent unicorn companies.

At OpenAI, we utilize an unsharded architecture with one writer and multiple readers, demonstrating that PostgreSQL can scale gracefully under massive read loads.

— PGConf.dev 2025, Bohan Zhang from OpenAI



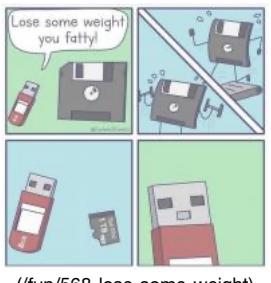
Bohan Zhang is a member of OpenAI's Infrastructure team. He studied under Professor Andy Pavlo (<https://www.cs.cmu.edu/~pavlo/>) at Carnegie Mellon University and co-founded OtterTune (<https://ottertune.com/>) with him.

Background

PostgreSQL serves as the core database supporting the majority of OpenAI's critical systems. If PostgreSQL experiences downtime, many of OpenAI's key services would be directly affected. There have been several instances in the past where issues related to PostgreSQL have led to outages of ChatGPT.

 (/admin.php?page_id=admin/fi)

RANDOM FUN



(/fun/568-lose-some-weight)

Lose some weight (/fun/568-lose-some-weight)

SUPPORT US

If you find this article helpful, please consider supporting our work.

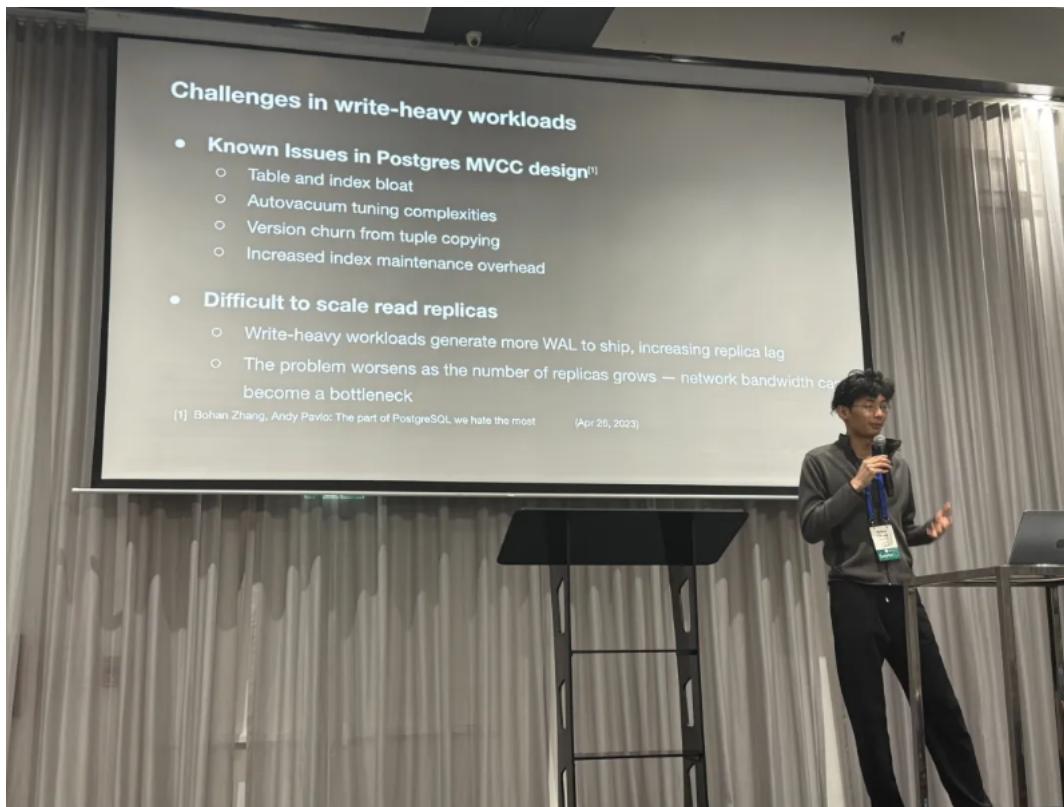
 DONATE (/donate.php)



OpenAI utilizes managed databases on Azure, employing a classic PostgreSQL primary-replica replication architecture without sharding. This setup consists of one primary database and over forty replicas. For a service like OpenAI, which boasts 500 million active users, scalability is a significant concern.

Challenges

In OpenAI's primary-replica PostgreSQL architecture, read scalability is excellent. However, "write requests" have become a major bottleneck. OpenAI has implemented numerous optimizations in this area, such as offloading write loads wherever possible and avoiding the addition of new services to the primary database.



PostgreSQL's Multi-Version Concurrency Control (MVCC) design presents some known issues, including table and index bloat. Tuning automatic garbage collection (vacuuming) can be complex, as each write operation generates a complete new version, and index access may require additional visibility checks. These design aspects pose challenges when scaling read replicas: for example, increased Write-Ahead Logging (WAL) can lead to greater replication lag, and as the number of replicas grows significantly, network bandwidth may become a new bottleneck.

Measures

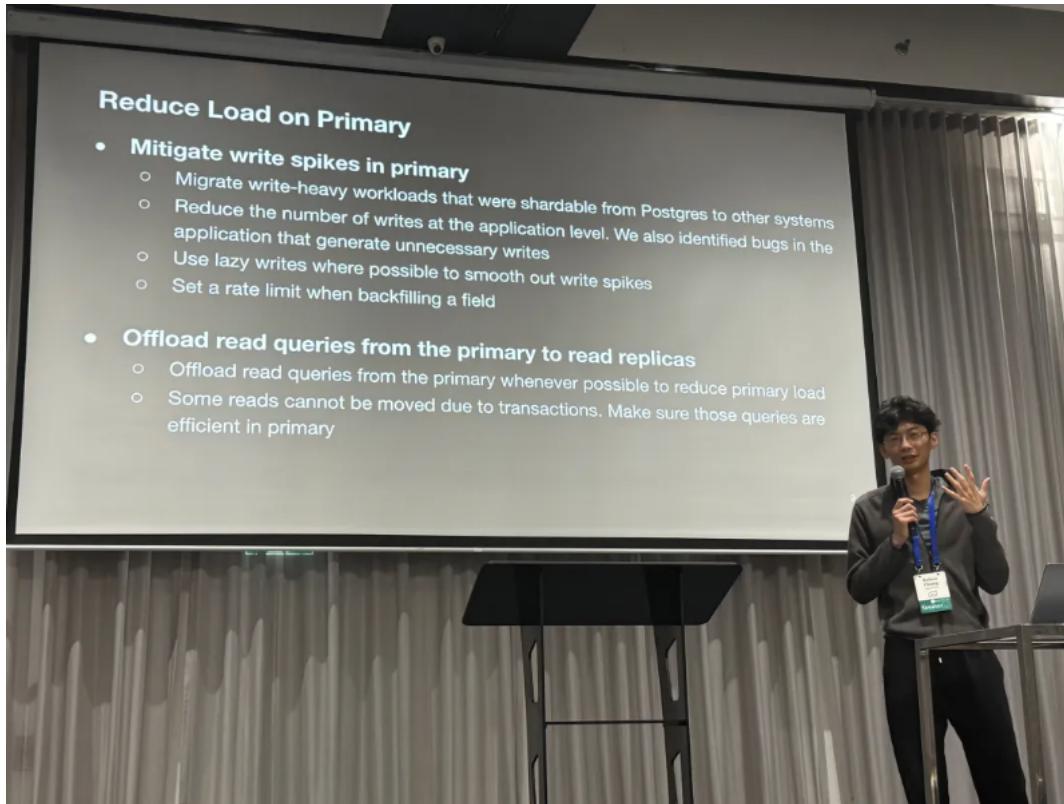
To address these issues, we have undertaken efforts on multiple fronts:

Controlling Primary Database Load

The first optimization involves smoothing out write spikes on the primary database to minimize its load. For example:

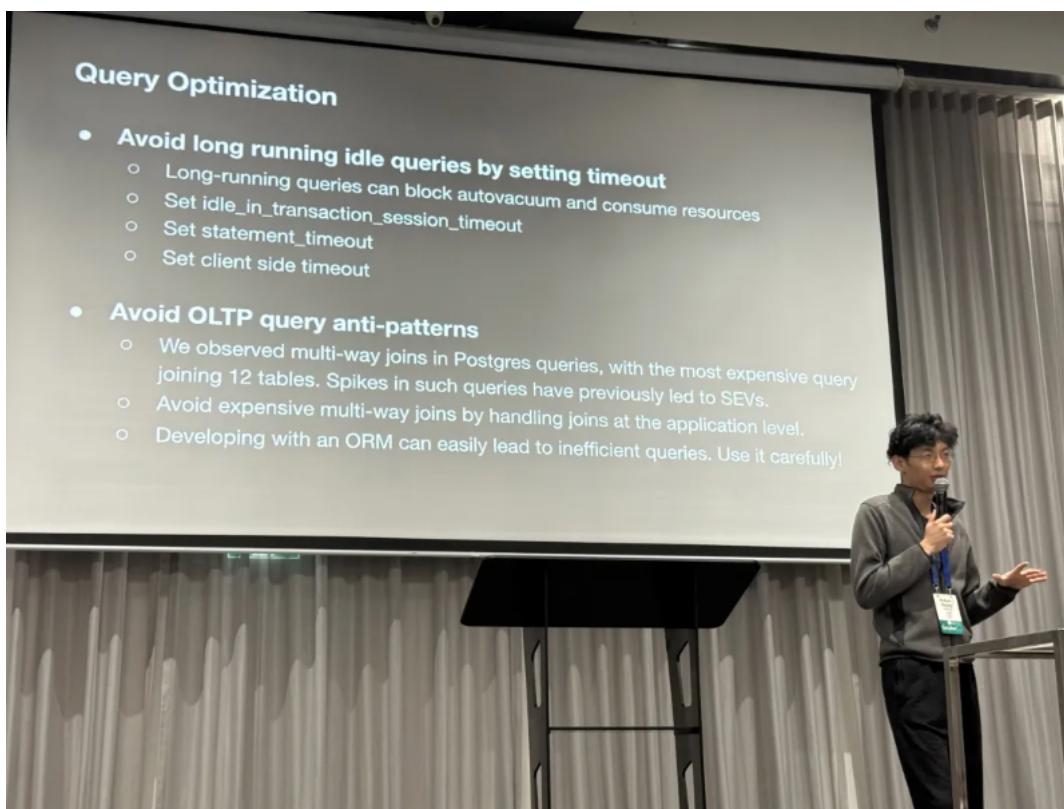
- Offloading all possible write operations.
- Avoiding unnecessary writes at the application level.
- Using lazy writes to smooth out write bursts.
- Controlling the frequency during data backfilling.

Additionally, OpenAI strives to offload as many read requests as possible to replicas. For read requests that cannot be removed from the primary database due to being part of read-write transactions, high efficiency is required.



Query Optimization

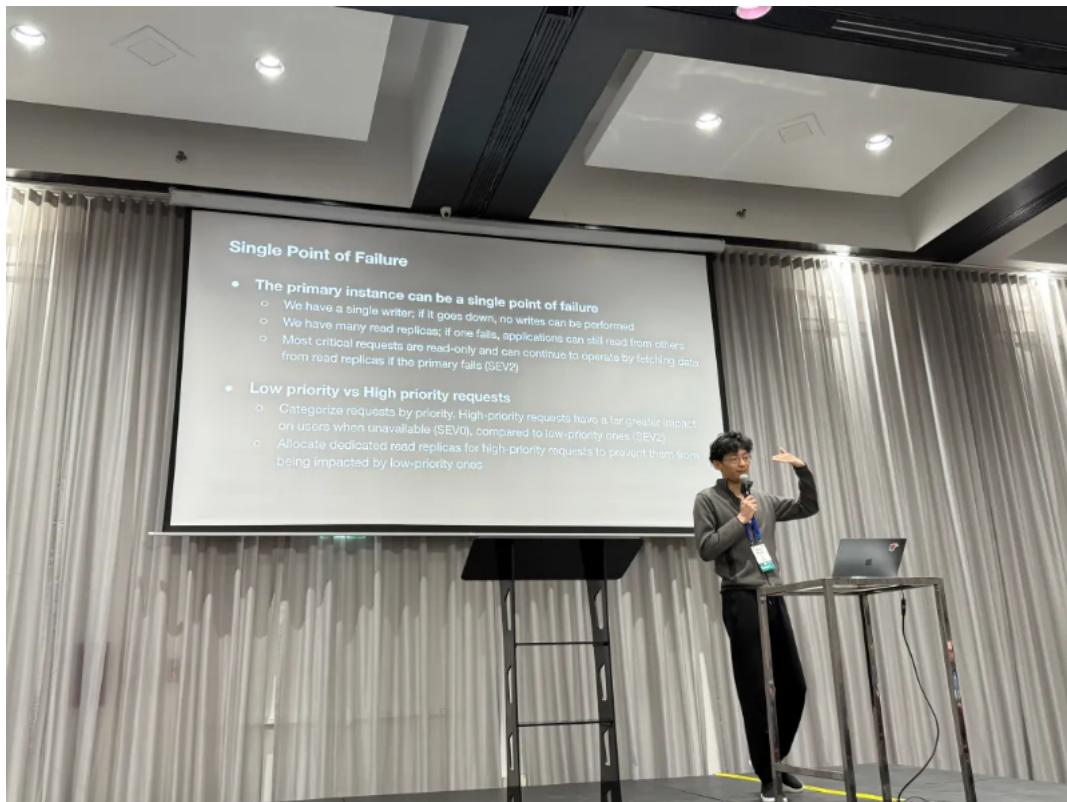
The second optimization focuses on the query layer. Since long transactions can hinder garbage collection and consume resources, timeouts are configured to avoid long “Idle in Transaction” sessions, with timeouts set at the session, statement, and client levels. Furthermore, complex multi-join queries (e.g., joining 12 tables at once) have been optimized. The presentation also specifically mentioned that using ORM can easily lead to inefficient queries and should be used cautiously.



Addressing Single Points of Failure

The primary database is a single point of failure; if it goes down, write operations cannot proceed. In contrast, we have many read-only replicas; if one fails, applications can still read from others. In fact, many critical requests are read-only, so even if the primary database fails, they can continue to read from it.

Moreover, we differentiate between low-priority and high-priority requests. For high-priority requests, OpenAI allocates dedicated read-only replicas to prevent them from being affected by low-priority requests.



Schema Management

The fourth measure is to only allow lightweight schema changes on this cluster. This means:

- Creating new tables or introducing new workloads is not permitted.
- Adding or removing columns is allowed (with a 5-second timeout), but any operation requiring a full table rewrite is not allowed.
- Creating or removing indexes is permitted but must use the CONCURRENTLY option.

Another issue mentioned is that long-running queries (>1s) during operation can continuously block schema changes, ultimately causing them to fail. The solution is to have the application optimize or offload these slow queries.

Results

- Scaled Azure-hosted PostgreSQL to handle over one million QPS (combined read and write) across the entire cluster, supporting OpenAI's critical services.
- Added dozens of replicas (approximately 40) without increasing replication lag.
- Deployed read-only replicas across different geographic regions while maintaining low

latency.

- Experienced only one PostgreSQL-related SEV0 incident in the past nine months.
- Reserved ample capacity for future growth.

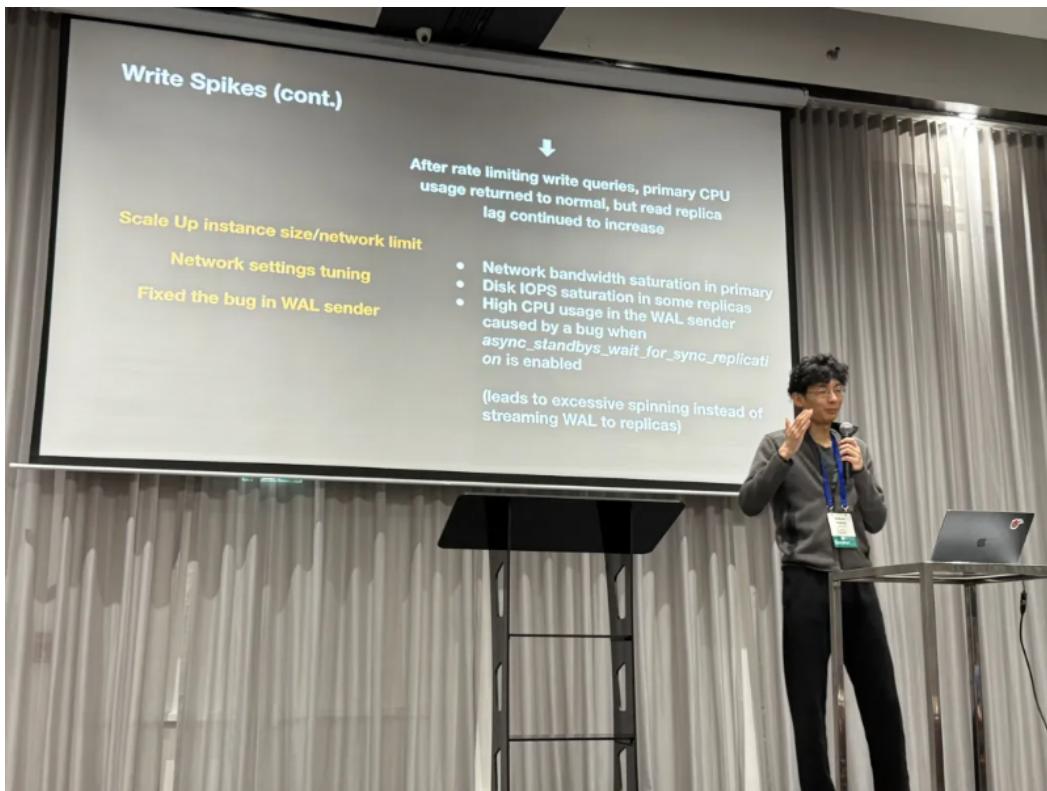
Incident Cases

OpenAI also shared several case studies of issues encountered:

- The first case involved a cache failure leading to a cascading effect.



- The second incident was particularly interesting: under extremely high CPU usage, a bug was triggered where, even after CPU levels normalized, the WALSender process continued spinning in a loop instead of properly sending WAL logs to replicas, resulting in increased replication lag.



Feature Requests

Finally, Bohan presented several issues and feature requests to the PostgreSQL developer community:

1. Regarding index management: unused indexes can lead to write amplification and additional maintenance overhead. OpenAI wishes to remove unnecessary indexes but, to minimize risk, they propose a “Disable” feature for indexes. This would allow monitoring performance metrics to ensure stability before permanently dropping the index.
2. On observability: currently, `pg_stat_statements` provides only average response times per query type, lacking direct access to p95 and p99 latency metrics. They hope for more metrics akin to histograms and percentile latencies.
3. Concerning schema changes: they desire PostgreSQL to record a history of schema change events, such as adding or removing columns and other DDL operations.
4. Monitoring view semantics: they observed a session with `state = Active` and `wait_event = ClientRead` persisting for over two hours. This indicates a connection remained active for an extended period post-QueryStart, and such connections cannot be terminated by `idle_in_transaction` timeouts. They seek to understand if this is a bug and how to address it.
5. Lastly, they suggest optimizing PostgreSQL’s default parameters, noting that the current default values are overly conservative. They inquire whether better defaults or heuristic-based settings could be implemented.

Lao Feng's Comments

Although PGConf.Dev 2025 primarily focuses on development, there are often user-side use case shares as well—like OpenAI’s scalability practices with PostgreSQL. Topics like this are actually quite interesting to core developers, since many of them have no concept of how PostgreSQL is used in extreme real-world scenarios.

Since the end of 2017, Lao Feng managed dozens of PostgreSQL clusters at Tantan, which was one of the largest and most complex deployments in China’s internet sector at the time: dozens of PostgreSQL clusters handling around 2.5 million QPS. Back then, their largest core cluster used a master with 33 replicas and carried around 400,000 QPS. The bottleneck was also on single-node write performance, which they eventually addressed through database and table sharding on the application side.

You could say that the issues encountered and the solutions applied in OpenAI’s talk were all things they’ve dealt with before. Of course, what’s different now is that today’s top-tier hardware is way more powerful than it was eight years ago. That allows a startup like OpenAI to use a single PostgreSQL cluster—with sharding or partitioning—to serve their entire business. This undoubtedly serves as another strong piece of evidence for the idea that “distributed databases are a false need.”

OpenAI uses managed PostgreSQL on Azure, with top-tier server specs. The number of replicas reaches over 40, including some cross-region replicas. This massive cluster handles around 1 million QPS (read + write) in total. They use Datadog for monitoring, and their services access the RDS cluster through application-side PgBouncer connection pooling from within Kubernetes.

Since OpenAI is a strategic-level customer, the Azure PostgreSQL team provides very hands-on support. But clearly, even with top-tier cloud database services, users still need strong awareness and capabilities on the application and operations side. Even with the brainpower of OpenAI, they still run into pitfalls in PostgreSQL operations in practice.

High availability wasn’t discussed in this talk, so we can assume that’s handled by Azure PostgreSQL RDS. Meanwhile, monitoring is critical for system ops. OpenAI uses Datadog to monitor PostgreSQL—and even with OpenAI’s financial resources, they still feel that Datadog is ridiculously expensive.

After the conference, during the evening social event, Lao Feng had a long chat into the early hours with Bohan and two other database founders. The private conversation was very engaging, though Lao Feng couldn’t reveal more details—haha.



Lao Feng Q&A

Regarding the issues and feature requests raised by Bohan, Lao Feng offers some answers here. In fact, most of the functionality OpenAI is looking for already exists within the PostgreSQL ecosystem—it just might not be available in the core PostgreSQL or on Azure RDS.

On Disabling Indexes

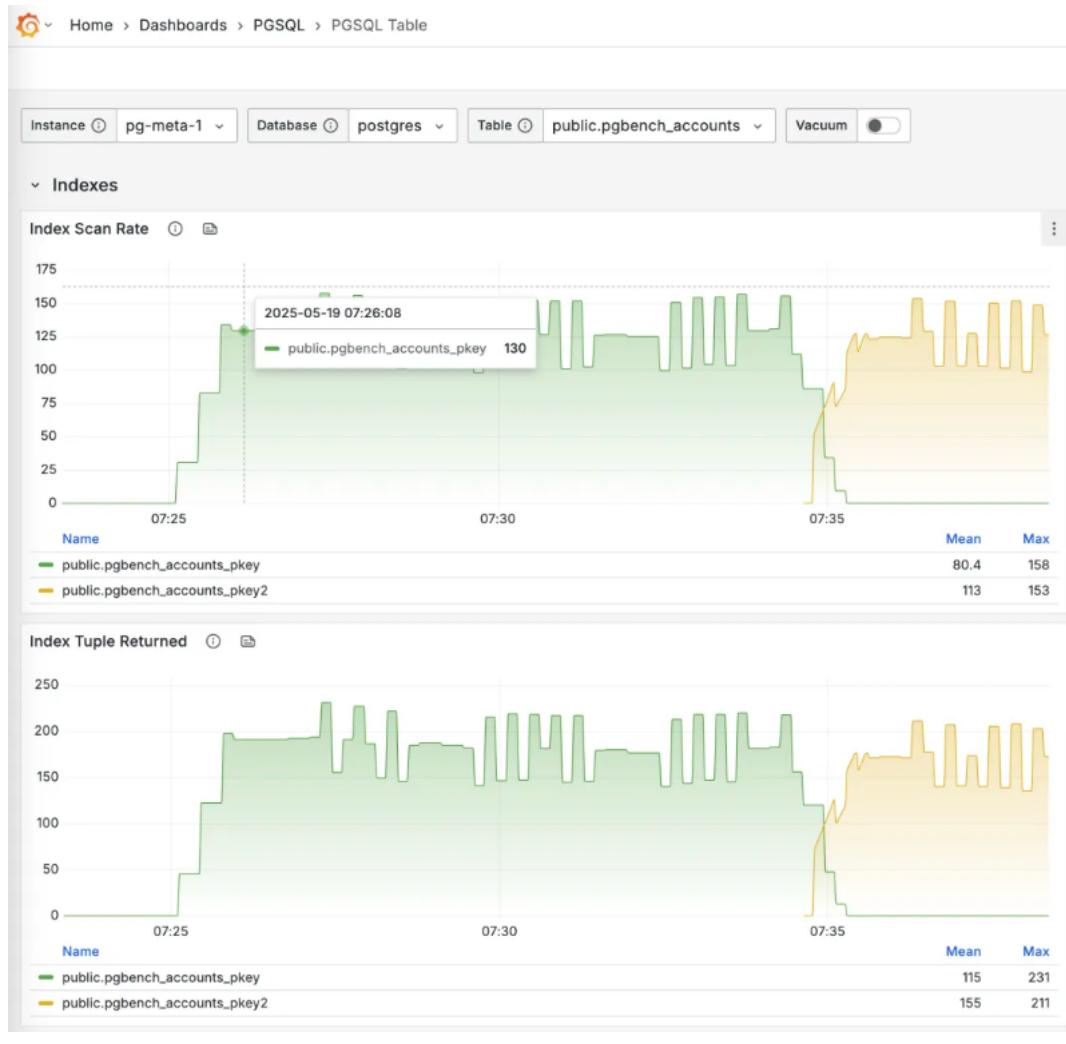
PostgreSQL actually does have a feature to disable indexes. You can simply set the `invalid` field to false in the `pg_index` system catalog. This makes the planner ignore the index, although it will still be maintained during DML operations. From a technical standpoint,

this is totally fine—this is the same mechanism used during concurrent index creation via the `isready` and `isvalid` flags. It's not black magic.

That said, it's understandable why OpenAI can't use this method—RDS doesn't grant superuser permissions, so you can't modify system catalogs directly to achieve this.

But going back to the original goal—avoiding accidental deletion of indexes—there's a simpler solution: just confirm via monitoring views that the index is not being used on either primary or replicas. If it hasn't been accessed for a long time, it's safe to delete.

Using the Pigsty monitoring system, you can observe the process of live index switching for PostgreSQL tables.



```
CREATE UNIQUE INDEX CONCURRENTLY pgbench_accounts_pkey2
ON pgbench_accounts USING BTREE(aid);

-- Mark the original index as invalid (won't be used) but still maintained
UPDATE pg_index SET indisvalid = false
WHERE indexrelid = 'pgbench_accounts_pkey'::regclass;
```

On Observability

`pg_stat_statements` likely won't provide P95 or P99 percentile metrics anytime soon, as this would drastically increase the memory footprint of the extension—maybe dozens of times.

While modern servers could handle it, extremely conservative environments might not. I

asked the maintainer of pg_stat_statements about this and it's unlikely to happen. I also asked Jelte, the maintainer of pgbouncer, and such functionality is also unlikely in the short term.

But the issue can be addressed. First, the pg_stat_monitor extension does provide detailed percentile latency (RT) metrics and would certainly work, though you'll need to consider the performance overhead of collecting such metrics. A second option is using eBPF to passively collect RT metrics, and of course, the simplest way is to add query latency monitoring directly in the application's data access layer (DAL).

The most elegant solution might be eBPF-based side-channel collection, but since they're using Azure's managed PostgreSQL without server access, this option is probably off the table.

On Schema Change History

Actually, PostgreSQL logs already offer this capability—just set log_statement to ddl (or more verbosely, mod or all), and all DDL statements will be logged. The pgaudit extension provides similar capabilities.

But I suspect what they really want is not logs, but a system view that can be queried via SQL. In that case, another option is to use CREATE EVENT TRIGGER to log DDL events directly into a data table. The pg_ddl_histORIZATION extension provides a much easier way to do this, and I've already compiled and packaged this extension.

However, creating event triggers also requires superuser privileges. AWS RDS has some special handling that makes this possible, but Azure's PostgreSQL doesn't seem to support it.

On the Semantics of Monitoring Views

In OpenAI's example, State = Active means the backend process is still within the lifecycle of a single SQL statement—it hasn't sent a ReadyForQuery message to the frontend yet, so PostgreSQL still considers the statement "not yet finished." As a result, resources like row locks, buffer pins, snapshots, and file handles are still considered "in use." WaitEvent = ClientRead means the process is waiting for input from the client. When both appear together, a typical case is an idle COPY FROM STDIN, but it could also be due to TCP blocking or being stuck between BIND and EXECUTE. So it's hard to say definitively whether it's a bug—it depends on what the connection is actually doing.

Some might argue that waiting for client I/O should count as "idle" from a CPU perspective. But State tracks the execution state of the statement, not whether the process is actively using the CPU. A query can be in the Active state while not running on CPU (when WaitEvent is NULL), or it can be looping on CPU waiting for client input (i.e., ClientRead).

Back to the core issue—there are ways to address it. For example, in Pigsty, when PostgreSQL is accessed via HAProxy, the primary service has a maximum connection lifespan (e.g., 24 hours) set at the load balancer level. In more stringent environments, this

can be as short as one hour. This means connections exceeding the lifespan are terminated. Ideally, though, the client-side connection pool should proactively enforce connection lifetimes instead of being forcibly disconnected. For offline, read-only services, this timeout isn't needed—allowing for long-running queries that may last for days. This approach provides a safety net for cases where a connection is Active but waiting on I/O.

That said, it's unclear whether Azure PostgreSQL offers this kind of control.

On Default Parameters

PostgreSQL's default parameters are extremely conservative. For example, it defaults to just 256 MB of memory (and can be set as low as 256 KB!). The upside is that PostgreSQL can start and run in virtually any environment. The downside? I've seen a production setup with 1 TB of physical memory still running with the default 256 MB configuration... (Thanks to double buffering, it actually ran for quite a while.)

Overall, I think conservative defaults aren't a bad thing. This issue can be solved with more flexible dynamic configuration. Services like RDS and Pigsty offer well-designed heuristics for initial parameter tuning, which already solves this problem quite well. That said, this feature could still be built into PostgreSQL command-line tools—e.g., during initdb, the tool could auto-detect CPU, memory, disk size and type, and set sensible defaults accordingly.

Self-Hosting?

The real challenges in OpenAI's setup don't stem from PostgreSQL itself, but rather the limitations of using managed PostgreSQL on Azure. One solution would be to bypass those restrictions by using Azure or another cloud's IaaS layer to deploy self-hosted PostgreSQL clusters on local NVMe SSD instances.

In fact, Pigsty (<https://pigsty.io/>) was built by Lao Feng specifically to address PostgreSQL challenges at this scale—it's essentially a self-hosted RDS solution, and it scales well. Many of the problems OpenAI has encountered—or will encounter—already have solutions implemented in Pigsty, which is open-source and free.

If OpenAI is interested, I'd be happy to offer some help. That said, when a company is scaling as fast as they are, tweaking database infrastructure might not be a top priority. Fortunately, they've got some excellent PostgreSQL DBAs who can keep pushing forward and exploring these paths.

The article is authorized by Lao Feng (<https://x.com/RonVonng>) to translate and republish here. the original link is at

<https://mp.weixin.qq.com/s/ykrasJ2UeKZAMtHCmtG93Q>

(<https://mp.weixin.qq.com/s/ykrasJ2UeKZAMtHCmtG93Q>)

ADVICE

(/article.php?tag=422)

POSTGRES

(/article.php?tag=2725)

OPENAI

(/article.php?tag=6931)

 (<https://www.facebook.com/share.php?u=https://www.pixelstech.net/article/1747708863-openai%3a-scaling-postgresql-to-the-next-level>)  (<https://twitter.com/intent/tweet?url=https://www.pixelstech.net/article/1747708863-openai%3a-scaling-postgresql-to-the-next-level&via=PixelstechNet>)  (<https://service.weibo.com/share/share.php?url=https://www.pixelstech.net/article/1747708863-openai%3a-scaling-postgresql-to-the-next-level&title=OpenAI%3A+Scaling+PostgreSQL+to+the+Next+Level+via+%40Pixelstech>)  (<https://reddit.com/submit?url=https://www.pixelstech.net/article/1747708863-openai%3a-scaling-postgresql-to-the-next-level&title=OpenAI%3A+Scaling+PostgreSQL+to+the+Next+Level>)  (<https://www.linkedin.com/shareArticle?mini=true&url=https://www.pixelstech.net/article/1747708863-openai%3a-scaling-postgresql-to-the-next-level&title=OpenAI: Scaling PostgreSQL to the Next Level&source=https://www.pixelstech.net/article/1747708863-openai%3a-scaling-postgresql-to-the-next-level>)

RELATED

Change password of postgres account in Postgres (/article/1362383440-change-password-of-postgres-account-in-postgres)

First Touch on OpenAI API (/article/1731382762-first-touch-on-openai-api)

10 advice from Jack Ma on success (/article/1440924811-10-advice-from-jack-ma-on-success)

Hologres vs AWS Redshift (/article/1710660707-hologres-vs-aws-redshift)

What does a contemporary web developer need to know? (/article/1387542689-what-does-a-contemporary-web-developer-need-to-know)

Advice on improving your programming skills (/article/1392994721-advice-on-improving-your-programming-skills)

GitHub Copilot may generate code containing GPL code (/article/1682104779-github-copilot-may-generate-code-containing-gpl-code)

Australian software engineer got asked algorithm question when entering US (/article/1488648567-australian-software-engineer-got-asked-algorithm-question-when-entering-us)

A couple of tips for beginning programmers (/article/1403009209-a-couple-of-tips-for-beginning-programmers)

Good ways to build communities around a web product (/article/1394627779-good-ways-to-build-communities-around-a-web-product)

1 COMMENT

 yusufn

May 21, 2025 at 3:33 am

 Reply

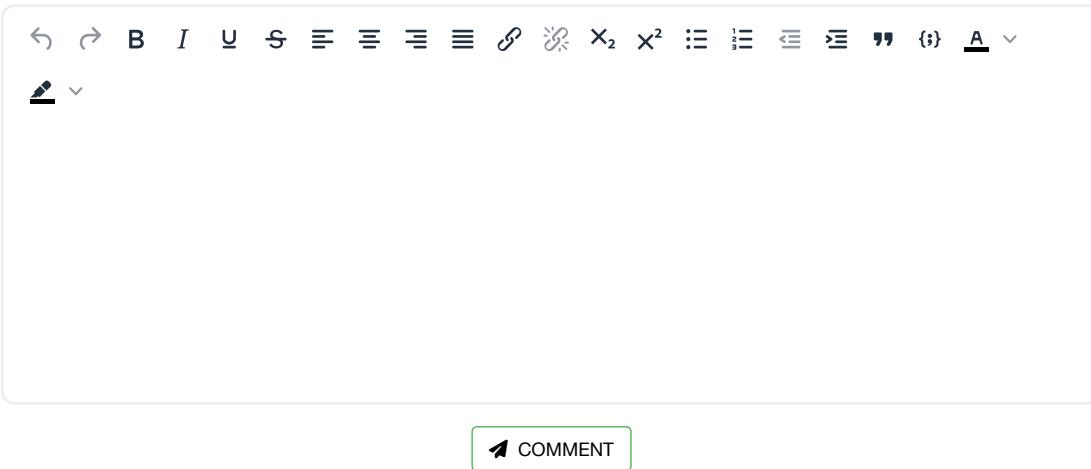
Index disabling is a long-awaited feature.



Anonymous



Get email when getting reply



ABOUT	HOW IT WORKS	FOLLOW US	FEEDBACK
About us (/document/aboutus.php)	Feeds (/article/1734054940-announcing-new-feed-service)	RSS (/feed.xml)	Contact us (/document/contactus.php)
Library (/documentopensource.php)	Post help (/document/article_post_help.php)	Weibo	Bug report (/bug_report.php)
Privacy policy (/documentprivacy.php)	Article guideline (/document/guideline.php)	Facebook	rightmod=1&wvr=5&mod=personinfo">Feedback (/feedback.php)
Terms of service (/documenttermofservice.php)	Submit article (/admin.php?page_id=admin/file_generator)	X	WeChat (/view/wechat.php)



©2025 PixelsTech.net All rights reserved