OpenAI

Scaling Postgres to the next level at OpenAI

Bohan Zhang
Member of Technical Staff, OpenAI
bohan@openai.com

Acknowledgements: Sicheng Liu, Chaomin Yu, Chenglong Hao, Dmitri Petrov, Kai Wang, Qi Xu, Jon Lee, Stas Llinskiy, Ben Ries, Venkat Venkataramani and many more at OpenAI infra team

Agenda

- Background: Postgres at OpenAI
- Optimizations: Scaling to millions of QPS in unsharded Postgres
- Case Study: Past Postgres Outages
- Feature Requests: Where PostgreSQL Could Do Better

About Myself

Member of Technical Staff @ OpenAI

↑

Cofounder @ OtterTune (CMU spin-off)

↑

Researcher @ Carnegie Mellon Database Group

Background: Postgres at OpenAI

Background

- **Postgres is the backbone of our most critical systems at OpenAI**
  - If Postgres goes down, many of our key features become unavailable
  - Postgres related incidents have had a significant impact to services like ChatGPT in the past

- **Scaling Postgres to meet OpenAI's demands is no trivial task**
  - We operated on a single primary instance in Azure without sharding for a long time
  - until we encountered write scalability limits…

Background

- **In a single-primary, multiple-replica architecture, write scalability remains a bottleneck**
  - Move write-heavy workloads that are shardable to other systems
  - New tables and workloads are not allowed
  - We did lots of optimizations to ensure the current architecture has sufficient runway to support existing read-heavy workloads and future growth

- **Postgres is not ideal for write-heavy workloads. But for OpenAI's read-heavy workloads, it can scale exceptionally well**

Challenges in write-heavy workloads

- **Known Issues in Postgres MVCC design**[1]
  - Table and index bloat
  - Autovacuum tuning complexities
  - Version churn from tuple copying
  - Increased index maintenance overhead

- **Difficult to scale read replicas**
  - Write-heavy workloads generate more WAL to ship, increasing replica lag
  - The problem worsens as the number of replicas grows — network bandwidth can become a bottleneck

[1] Bohan Zhang, Andy Pavlo: The part of PostgreSQL we hate the most (Apr 26, 2023)

Read-heavy workloads are still served by Unsharded Postgres in Azure

But How?

Why Postgres Remains Unsharded

- Shardable, write-heavy workloads have already been migrated to other systems.
- New tables are no longer allowed in Postgres. For feature additions that require new tables, use alternative systems.
- Sharding current workloads in Postgres is difficult due to the complexity of migrating hundreds of application endpoints.
- Current workloads are read-heavy, and with careful optimizations, the existing architecture has sufficient runway.
- Sharding is not a near-term priority but remains a possibility for the future.

Reduce Load on Primary

- **Mitigate write spikes in primary**
  - Migrate write-heavy workloads that were shardable from Postgres to other systems
  - Reduce the number of writes at the application level. We also identified bugs in the application that generate unnecessary writes
  - Use lazy writes where possible to smooth out write spikes
  - Set a rate limit when backfilling a field

- **Offload read queries from the primary to read replicas**
  - Offload read queries from the primary whenever possible to reduce primary load
  - Some reads cannot be moved due to transactions. Make sure those queries are efficient in primary

Query Optimization

- **Avoid long running idle queries by setting timeout**
  - Long-running queries can block autovacuum and consume resources
  - Set idle_in_transaction_session_timeout
  - Set statement_timeout
  - Set client side timeout

- **Avoid OLTP query anti-patterns**
  - We observed multi-way joins in Postgres queries, with the most expensive query joining 12 tables. Spikes in such queries have previously led to SEVs.
  - Avoid expensive multi-way joins by handling joins at the application level.
  - Developing with an ORM can easily lead to inefficient queries. Use it carefully!

Single Point of Failure

- **The primary instance can be a single point of failure**
  - We have a single writer; if it goes down, no writes can be performed
  - We have many read replicas; if one fails, applications can still read from others
  - Most critical requests are read-only and can continue to operate by fetching data from read replicas if the primary fails (SEV2)

- **Low priority vs High priority requests**
  - Categorize requests by priority. High-priority requests have a far greater impact on users when unavailable (SEV0), compared to low-priority ones (SEV2)
  - Allocate dedicated read replicas for high-priority requests to prevent them from being impacted by low-priority ones

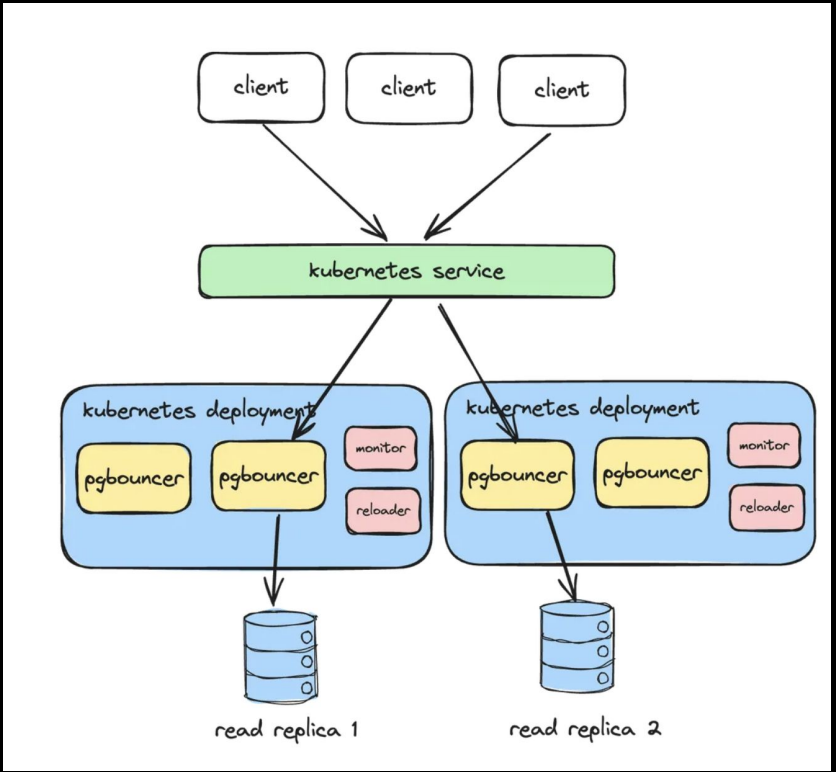
Rate Limit

- **A surge from a single expensive query can bring down the entire instance**
  - We had some expensive queries running on the primary (like 12-way joins), the volume was typically low
  - A sudden spike in one of these queries took down the entire instance

- **Rate Limiter**
  - Rate limit ***application-level functions*** to reduce load during peak traffic
  - Rate limit the creation of ***new connections*** to prevent connection pool exhaustion
  - Rate limit specified ***query digests*** to control the impact of expensive queries

Connection Pooling

- **PGBouncer as Postgres Proxy**
  - Acts as a connection pool, enabling connection reuse
  - Can significantly reduce connection latency (~5ms vs. 50ms)
  - Reduces the number of connections, which is important given the 5k connection limit on the primary
  - If a read replica fails, traffic is automatically rerouted to other available replicas



Schema Management

- **Only lightweight schema changes are permitted**
  - Creating new tables or introducing new workloads in Postgres is not allowed
  - We allow adding / removing columns in tables (with 5-seconds timeout). Any changes that require a table rewrite are not allowed
  - Indexes can be added or dropped concurrently

- **Schema changes can be blocked by consistent queries**
  - If long-running queries (e.g., >1s) are consistently present on the target table, the migration may get blocked and fail
  - Fix those queries in applications, or move them to read replicas
  - SELECT * FROM pg_stat_activity WHERE query like '%table_name%' and now() - query_start > interval '1 seconds'

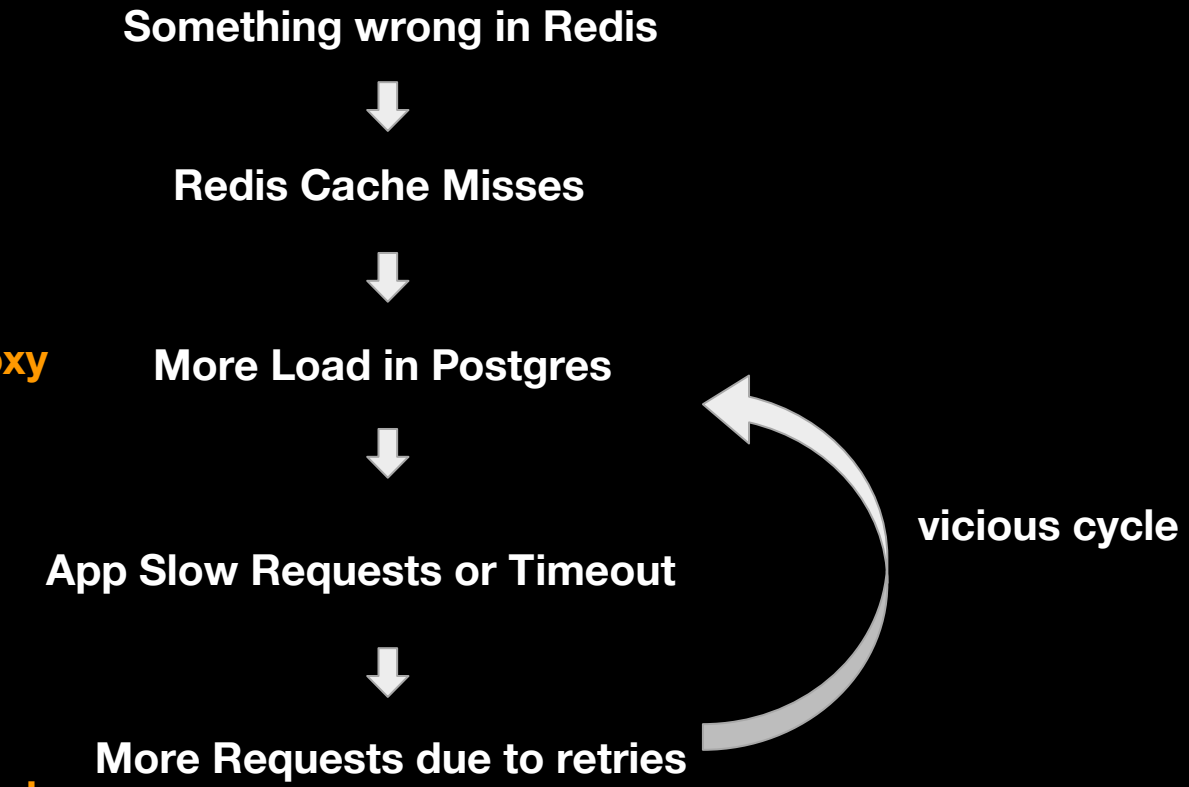
Results

- **Scaled Azure PostgreSQL to millions of QPS, powering OpenAI's critical services**
- **Added dozens of read replicas with no increase in replication lag**
- **Maintained low-latency across geo-distributed read replicas**
- **Only one SEV-0 incident involving PostgreSQL in the past 9 months since I joined OpenAI**
- **Sufficient capacity headroom to sustain future growth**

Huge thanks to the Azure PostgreSQL team for their unwavering support along this journey!

Case Study: PostgreSQL Outage at OpenAI

Cache Misses



Rate limit on Postgres and proxy

Rate limit on App;
Drop requests when overloaded;

Expensive Queries

Postgres primary has some *expensive* multi-way join queries

Optimize expensive queries

Allow blocking/rate limiting specific queries

A sudden *increase* in volume of the expensive query (e.g., new features deployed + retries)

Postgres primary high cpu usage (95%+)

Avoid using the primary for read-only requests to prevent a single point of failure

Postgres primary queries become super slow, impact critical service

Write Spikes

Optimize applications to reduce writes

Move read queries out of primary if possible

Remove unused indexes to reduce write amplification

A huge spike in writes

High CPU usage in primary (90%+)

Increased Read Replica Lags (> 10 mins)

Queries become very slow,
Read replica queries become stale,
New writes are not accepted,

impact ChatGPT services

Write Spikes (cont.)

After rate limiting write queries, primary CPU usage returned to normal, but read replica lag continued to increase

Scale Up instance size/network limit

Network settings tuning

Fixed the bug in WAL sender

- Network bandwidth saturation in primary
- Disk IOPS saturation in some replicas
- High CPU usage in the WAL sender caused by a bug when *async_standbys_wait_for_sync_replication* is enabled

(leads to excessive spinning instead of streaming WAL to replicas)

Where PostgreSQL Could Do Better ?

Observability

- **Query latency**
  - pg_stat_statement only provides average latency per query digest
  - We cannot get query percentiles (like p95, p99) directly
  - We hope it can have more information like histogram and percentile latency

- **Schema changes**
  - It would be valuable for PostgreSQL to store schema change events, including operations like adding/dropping columns or indexes, and other DDL

Disable index

- **Unused indexes** increase maintenance cost and write amplification
  - We want to drop unused indexes
  - However, to minimize risk, we prefer to **disable** indexes temporarily and monitor performance **before** dropping them permanently.

- **Current limitation:** PostgreSQL does not support disabling indexes.
  - Drop the index to stop its use, and recreate it manually if needed as fallback
  - Recreating large indexes can take long time

Long running active query?

- We found some **active** queries that have been waiting on the client for a very long time.

Sampled State:	active
In-flight Duration:	2h 23m

The query is **active** for more than 2 hours

wait_event	ClientRead
wait_event_group	Network
wait_event_type	Client

The wait event is **ClientRead**, indicating the query is waiting for a request from the client.

query_start	2025-05-11T07:56:50
state	active
state_change	2025-05-11T07:56:50

query_start ~= state_change, indicating the query has been in a ClientRead wait state the entire time.

State is active; cannot be killed by idle_in_transaction_session_timeout

Is it a bug in Postgres? Should the state be idle_in_transcation?
If not, how to kill it automatically?

Automatic Knob Tuning

- **Postgres default knob values are notoriously bad**
  - Better default knob values (heuristic-based)
  - Adaptive knob tuning based on workloads (like autovacuum)

- **Industry examples:**
  - Default knob values are much better on managed cloud providers like AWS RDS and Azure Database for PostgreSQL.
  - Adaptive autovacuum tuning in Google AlloyDB

At OpenAI, we've proven that PostgreSQL can scale to support massive read-heavy workloads - even without sharding - using a single primary writer

If you are a developer, or building a startup, start with Postgres (for read-heavy workloads)

Thank you