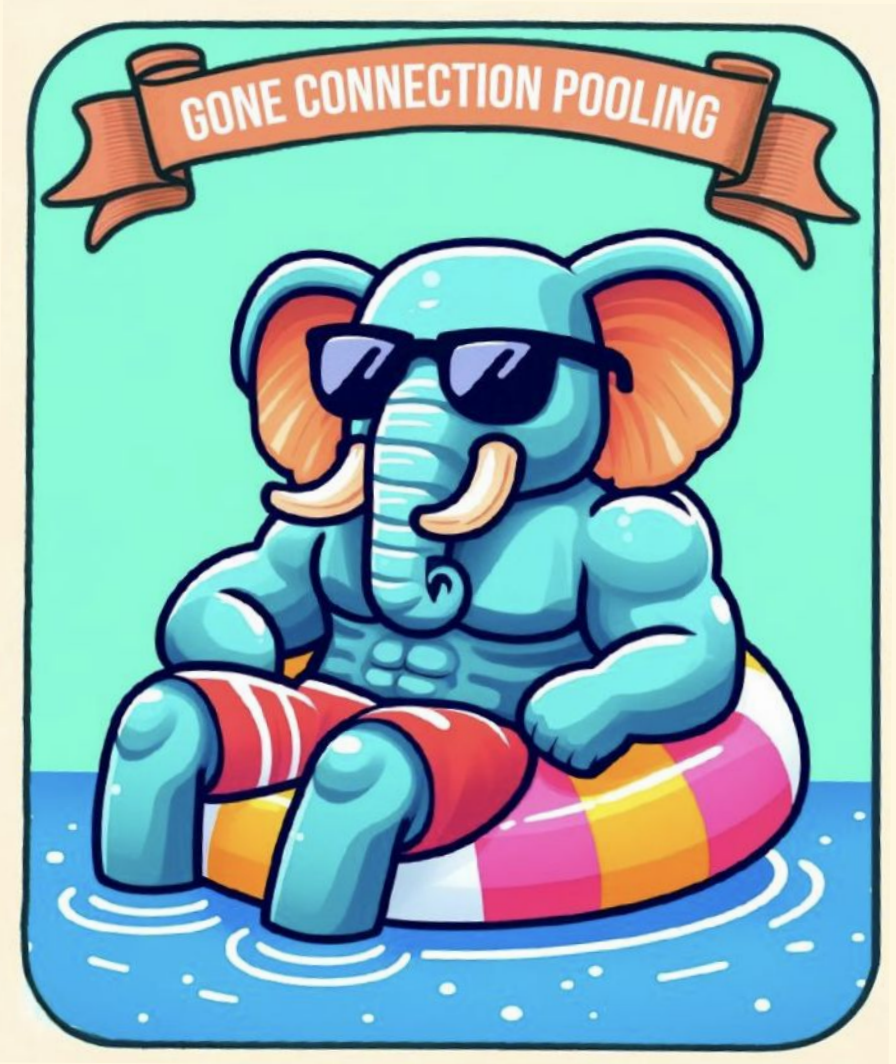
Scaling Beyond 100K Connections Using In-Database Pooling

Learnings from connection management and operational complexity in Postgres environments.

Jonah H. Harris

PGConf NYC
2024



nextgres.com
info@nextgres.com

NEXTGRES

AGENDA

Today’s session is focused on the importance of connection pooling and operational complexity in Postgres environments.

- The problem.
- The current solutions.
- A case study.
- The future.
- Q&A and closing remarks.

INTRODUCTION

I started developing database compatibility solutions out of necessity, evolving my expertise into a 20+ year career across various systems.

CEO

Jonah Harris

linkedin.com/in/jonahharris/

28 years in software development, 25 years in database internals and compatibility, led 100+ member startup and public technology organizations, Oracle & Postgres expert.

CTO @ MariaDB (NYSE)

CTO & Dir. AI/ML @ The Meet Group (NASDAQ)

Founding Engineer @ EDB

1999

The first version of NEXTGRES was created, compatible with Oracle Database.

NEXTGRES

2004

Presented a demo of Ingres r3 with support for PL/SQL and a SQL*Net compatible front-end at The Ingres Conference.

INGRES

2005

Joined EDB, licensed some IP, worked on compatibility, performance, and joint project to add PL/SQL to IBM DB2.

EDB

2014

Developed mongrel.io, a proxy compatible with MongoDB.

mongrel.io
Mongo + Relational

2024

Refactoring to incorporate the latest Postgres advancements.

THE HISTORY

Databases historically manage connections using events, threads, processes, or a combination of these approaches, each with varying impacts on performance and resource efficiency in high-concurrency environments.

THE CHALLENGE

Postgres employs a process-per-connection model, spawning a separate process for each client connection.

THE PROBLEM

Postgres environments tend to suffer from excessive resource consumption and latency due to inefficient connection handling.

THE CONSEQUENCE

Without efficient connection management, these systems experience resource exhaustion, increased latency, and difficulty scaling to meet demand.

THE DISCOVERY

Connection pooling optimizes resource usage and reduces latency by reusing existing connections, efficiently managing high-concurrency environments.

THE SOLUTION

Connection poolers were built to alleviate these issues, reducing the overhead of managing client processes.

THE USUAL SUSPECTS

Several Postgres-specific connection poolers emerged to address this bottleneck, and they're still actively developed.

- PgBouncer
- Pgpool-II
- PgCat
- Odyssey
- Supavisor
- pgagroal

PGBOUNCER

PgBouncer focuses on being lightweight and simple, ideal for efficient transaction and session pooling.

- Created by Marko Kreen and used at Skype.
- Actively developed and maintained by PostgreSQL community contributors.
- Ideal for lightweight environments needing fast transaction and session pooling.
- Not suited for complex use cases requiring features like replication or query caching.

PGPOOL-II

Pgpool-II offers a feature-rich solution with query caching, replication, and load balancing for complex environments.

- Created by Tatsuo Ishii.
- Actively developed and maintained by the Pgpool Development Group.
- Best for feature-rich environments needing replication, load balancing, and query caching.
- Can introduce complexity and overhead, making it unsuitable for simple, low-latency setups.

PGCAT

PgCat is designed for sharding and failover, making it perfect for large-scale, distributed applications.

- Created by PostgresML.
- Actively developed and maintained by PostgresML and community contributors.
- Perfect for sharded, distributed systems needing query routing and failover.
- Not ideal for small or simple deployments without sharding or high-availability needs.

ODYSSEY

Odyssey delivers high performance, built to handle high-concurrency environments with ease.

- Created by Yandex.
- Actively developed and maintained by Yandex and community contributors.
- Optimized and best suited for environments requiring multithreading and heavy concurrency.
- May be overkill for simple or low-concurrency workloads.

SUPAVISOR

Supavisor excels at real-time connection pooling, tailored specifically for multi-tenant architectures.

- Created by Supabase.
- Actively developed and maintained by Supabase and community contributors.
- Ideal for SaaS platforms and multi-tenant systems with real-time data needs.
- Overkill for single-tenant environments or cases without real-time requirements.

PGAGROAL

pgagroal emphasizes low-latency connection management with minimal overhead for resource-sensitive deployments.

- Created by Red Hat.
- Actively developed and maintained by Red Hat and community contributors.
- Designed for low-latency environments where minimal overhead is critical.
- Lacks the advanced features of larger poolers, so not recommended for complex systems.

Connection poolers are relatively easy to use,
but are a PITA to manage.

CASE STUDY

The Meet Group

THE TWO BIGGEST PROBLEMS

Postgres' process-per-connection model results in scalability issues primarily in two common scenarios.

- Short Lived Connections
- High Concurrency

SHORT LIVED CONNECTIONS

Short-lived connections create excessive resource overhead and increase latency.

- Each new connection spawns a separate process, leading to high resource overhead even for brief interactions.
- Rapidly opening and closing connections results in unnecessary CPU and memory consumption.
- The constant creation and teardown of processes adds significant latency, especially in high-traffic environments.

HIGH CONCURRENCY

High concurrency overwhelms resources, leading to poor scalability and performance.

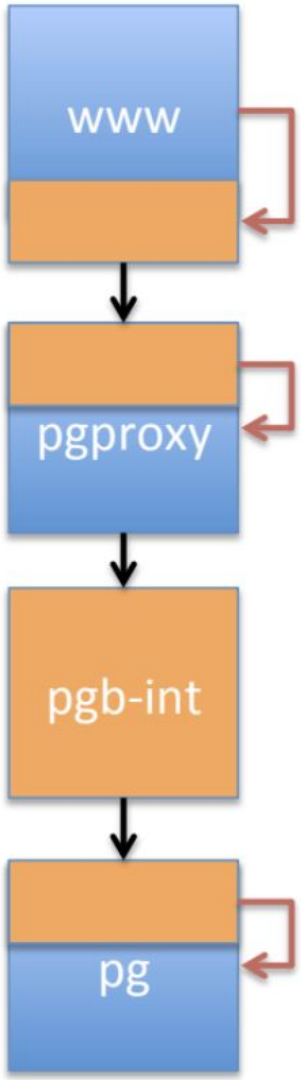
- Spawning a process for every connection quickly exhausts system resources as connection numbers grow.
- Memory and CPU usage scale poorly in environments with thousands of simultaneous connections.
- Managing large numbers of processes increases system complexity and can degrade performance under heavy load.

THE IMPORTANT QUESTION

How did this happen?

OUR ENVIRONMENT

PgBouncer all the way down…



OUR ADVANCEMENT

Multi-threading addressed PgBouncer's single-threaded limitations for better scalability on the pgbouncer-int tier.

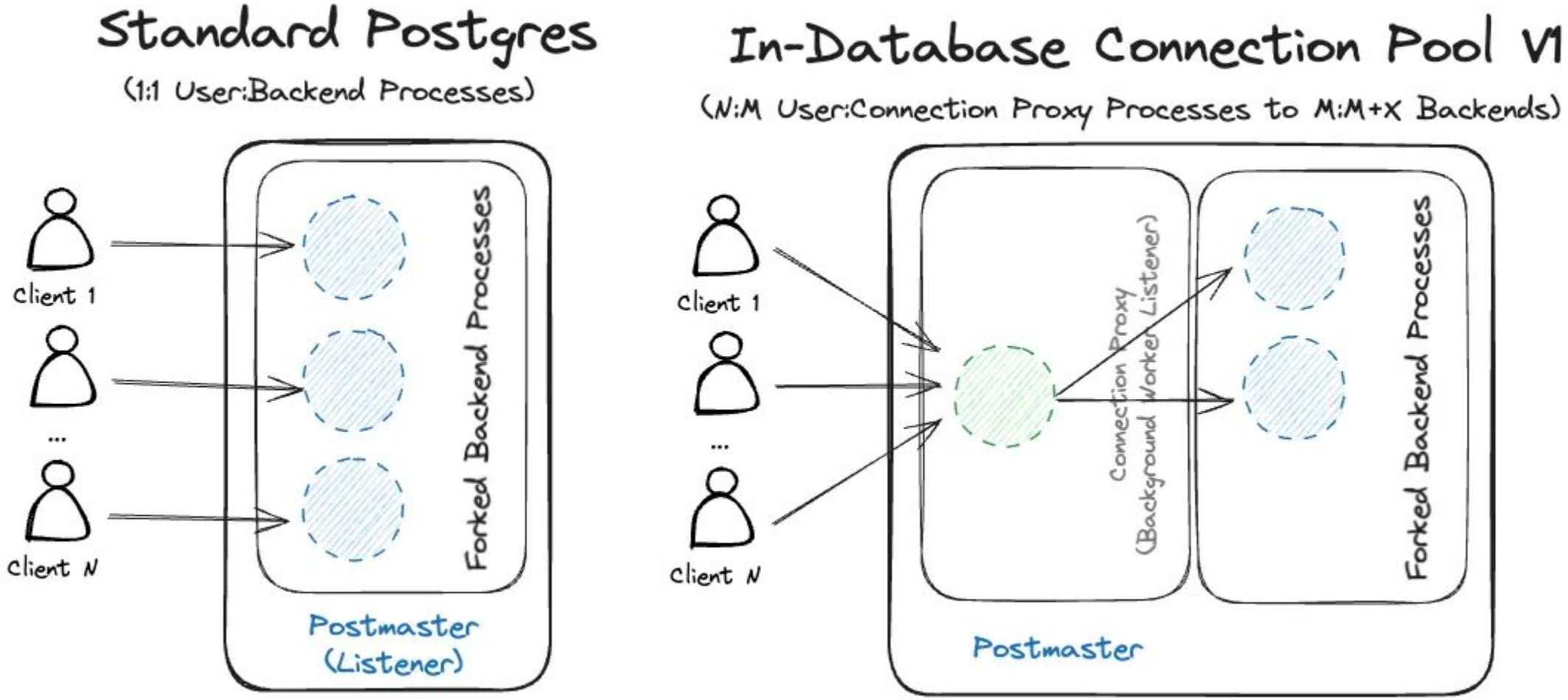
- Allowed it to handle more connections and higher throughput.
- Workloads were distributed across multiple CPU cores.
- Wait times for connections were reduced, lowering latency.

This worked well, but…

It was an operational management nightmare.

OUR PEERS

Postgres is one of the only remaining databases employing a process-per-connection model; others have solved it for a reason.



OUR GOAL

Our goal was to reuse connections to improve resource efficiency and reduce overhead within the database.

- Reuse existing connections and reduce the need to spawn new processes to conserve memory and CPU resources.
- Lower memory and CPU usage by minimizing the number of active processes and connections.
- Distribute incoming connection requests intelligently to optimize performance.

OUR FIRST PASS

We first embedded PgBouncer as a background worker extension to provide connection proxy architecture functionality.

- I know it sounds crazy…
- It was not the right approach…

OUR PEERS

PgBouncer as a background worker-based proxy efficiently managed incoming connection requests, but was difficult to debug..

- We switched to Konstantin Knizhnik's connection proxy patch and refactored it as a bgworker.
- It also conserved resources and efficiently managed connections.
- Minimizes resource consumption through connection routing at the proxy layer.
- Route incoming connections to the most suitable database session and reduce new process creation.

OUR PEERS

We started reincorporating PgBouncer elements to provide additional capabilities and enhance native performance.

- Utilize a background worker-based proxy to manage and route connections, reducing the need for new processes.
- Incorporates PgBouncer's features (like prepared statements) directly into the extension for enhanced capabilities.
- Simplify deployment by building it as a Postgres extension with no external dependencies.

OUR PEERS

We can achieve parallelism without the complexity of traditional multithreading, but…

- Use SO_REUSEPORT to split connection proxying across multiple background worker processes for Linux/UNIX environments.
- Peering
- We still think multi-threading is right here…

OUR PEERS

This extension boosts PostgreSQL's performance and scalability for high-concurrency environments.

- Reduce the delay in connection setup and enable better use of server resources for critical database operations.
- Support more concurrent connections and allow larger applications and services to scale without degrading performance or requiring an intermediary.
- Optimize resource usage to cut operational expenses and improve server efficiency.

OUR PEERS

There are still a number of limitations with this approach.

- Connection reuse complicates the handling of temporary tables, which are tied to individual sessions.
- Per-session features such as session variables and advisory locks may not behave as expected across pooled connections.
- Since the extension operates outside the core backend, certain session-bound operations cannot be fully managed within the pooler.

RECAP

Wrapping Up and Key Takeaways

- Connection management in Postgres remains an existing problem to solve for *some* environments.
- Connection pooling solves *both* short-lived and high concurrency connection cases.
- It's hard to beat a multi-threaded PgBouncer for *most* things.
- In-database connection pooling *eliminates* complex deployments.

Q&A

Let's Jump Into Your Questions

CONTACT

Let’s Connect

https://github.com/nextgres/nextgres-idcp

https://github.com/nextgres/nextgres-idcp-pgb

jonah@nextgres.com