COMPARING CONNECTION POOLERS FOR POSTGRESQL

JULIAN MARKWORT
Senior Database Consultant

pgconf.eu 2024

Introduction

Julian Markwort
Senior Database Consultant

CYBERTEC
POSTGRESQL SERVICES & SUPPORT

- PostgreSQL Consulting
- PostgreSQL Support
- PostgreSQL Remote DBA
- and more ...

ohne Cybertec Polo/Shirt!

Introduction

What is a Pooler?

- a transparent middleware…
- that receives (more) connections from clients…
- and attaches them to (fewer) connections to a server

Why to use Poolers?

Mostly because of bad applications or architecture

Problem: Too Many Connections

application runs on dozens of servers, every instance opens up hundreds of database connections

- you need to increase `max_connections` to over 9000!

Problem: Too Many Connections

many connections can result in performance regression

- even if they are mostly idle (some improvements in PG >= 14)
- 2MB of memory used for every connection
- lots of functions iterate over all connections (e.g. check active transactions)

Problem: Short Connection Lifetime

application opens a new database connection for every request and closes it afterwards

- you don’t want to lose track of your connections
  - open them only when necessary
  - close them as fast as possible

Problem: Short Connection Lifetime

new connections to PostgreSQL are expensive

- a new *session* is `fork()`ed
- round trips to establish TLS
- authentication needs to happen
- session is tied into shared memory
- load `relcache` into session

Problem: Long Connection Lifetime

application opens up a bunch of connections and uses them forever

- you don’t want to lose track of your connections
  - so you open many connections when the application starts
  - and stop all connections them when the application stops

Problem: Long Connection Lifetime

- sessions can acquire cache bloat (metadata such as table definitions, etc.)
- your application will probably need a restart when the database crashes

Problem: Too Many Read-Only Queries

application does a lot of reads and only very few writes

- you start scaling up
- everything goes to a single PostgreSQL instance
- at some point you can’t buy more CPU cores

Problem: Too Many Read-Only Queries

a single PostgreSQL instance might not keep up with your demands

- how can you use more instances without changing the application?

Problem: Low Connection Utilization

connections are certainly going to be idle for some % of time

- application needs to process results before sending next query
- network latencies are usually unavoidable

Problem: Low Connection Utilization

less utilization means you need more connections to saturate throughput

- at 5% utilization you would need 20 times as many connections
- more connections in PostgreSQL means more context switches

Problem: Thundering Herd

your application is running great with over 9000 connections to PostgreSQL

- some day there is some little problem
  - maybe a long running transaction results in bloat and longer run times
  - or some minor maintenance cannot acquire a lock fast enough
- over 9000 connections are now jammed, waiting to be executed
  - all of them start thrashing the database

Problem: Thundering Herd

you can probably get the same throughput with a pooler

- you keep the *congestion* in a wait queue in the pooler
- since the pooler doesn't need to also compute costly queries, it can concentrate on scheduling

Challenges for Poolers

- misconfiguration can lead to downtime
- misconfiguration (`pool mode = transaction`) can lead to unexpected behaviour and broken applications
  - prepared statements
  - temporary tables
  - advisory locks
  - (anything that crosses transaction boundaries)

Challenges for Poolers

- authentication needs to be managed in pooler (in addition to database)
- more things to troubleshoot (timeouts, connections breaking, authentication issues)

Drawbacks of Poolers

- overhead in poolers
- additional latency through additional network hops
- reduced visibility into origin of connections to PostgreSQL
- PostgreSQL error messages (e.g. authentication) are not always relayed to clients

Drawbacks of Poolers

- since poolers are by design transparent, you cannot detect easily if you are connected to one
- some tasks still need to bypass the pooler
  - maintenance
  - pg_dump, pg_basebackup
  - replication

Comparing Poolers

Overview

- pgpool-II (released 2006)
- pgbouncer (2007)
- odyssey (2019)
- pgagroal (2019)
- pgcat (2022)
- supavisor (2023)

too complicated

→ no additional features

→ Yandex, unstable

→ TLS pooler → server not supported (WIP)

→ lacks scram-sha-256 (on the roadmap)

incomprehensible setup

pgpool-II

Clustering tool for PostgreSQL

- successor to pgpool (release 2003)
- started by Tatsuo Ishii at SRA OSS (Japan), now the project is owned by the Pgpool Global Development Group
- pgpool-II License (“similar to BSD and MIT”)
- written in C99
- primarily a pooler, but with lots of additional features

pgpool-II

- primarily a pooler, but with lots of additional features:
  - various replication modes
  - load balancing
  - failover
  - routing of read-write and read-only queries
  - query cache
- due to plethora of features (especially replication), configuration is complex

pgbouncer

Lightweight connection pooler for PostgreSQL

- started by Marko Kreen at Skype in 2007
- ISC License (“basically a *stripped down* version of the MIT and simplified BSD licenses”)
- written in C99
- only a pooler, not much more

odyssey

Scalable PostgreSQL connection pooler

- started by Dmitry Simonenko at Yandex in 2016, released version 1.0 in 2019
- BSD-3-Clause license
- written in C99
- similar to pgbouncer, but with additional features and multi-threading

pgagroal

High-performance protocol-native connection pool for PostgreSQL

- started by Jesper Pedersen at RedHat in 2019
- BSD-3-Clause license
- written in C17
- a pooler written from scratch

pgcat

PostgreSQL pooler with sharding, load balancing and failover support

- started by Lev Kokotov at PostgresML in 2022
- MIT license
- written in Rust
- similar to pgbouncer, but with additional features and multi-threading

supavisor

Scalable, cloud-native Postgres connection pooler

- started by Stas “abc3” at supabase in 2023
- Apache-2.0 license
- written in Elixir
- incomprehensible setup (to me)

Comparison

Authentication

	pgbouncer	pgagroal	pgcat	odyssey
auth query	y	y	y	y
md5	y	y	y	y
scram-sha-256	y	y	n	y
client cert	y	n	n	y

TLS

	pgbouncer	pgagroal	pgcat	odyssey
client -> pooler	y	y	y	y
pooler -> server	y	WIP	y	y

Features

	pgbouncer	pgagroal	pgcat	odyssey
live config reload	y	y	y	y
pause	y	n	y	n
parallelism	socket reuse	y	y	y

possible to hide restarts / failovers

Features

	pgbouncer	pgagroal	pgcat	odyssey
prepared statements	y	n	n	y
read-only	n	n	y	n
load balancing	n	n	y	y
sharding	n	n	y	n

Pooling

	pgbouncer	pgagroal	pgcat	odyssey
session	y	y	y	y
transaction	y	y	y	y
statement	y	n	n	n
limit lifetime	y	n	y	y

Monitoring

	pgbouncer	pgagroal	pgcat	odyssey
pgbouncer-like	y	n	y	y
prometheus	via exporter	y	y	y

Miscellaneous

	pgbouncer	pgagroal	pgcat	odyssey
documentation	*****	***	**	*
stable	*****	***	***	**
PGDG apt repo	y	y	n	n
PGDG yum repo	y	y	n	n
apt repo			y	

Which one to use?

a lot of personal opinion, mostly based on gut feeling

pgbouncer

use pgbouncer. “Often imitated, never duplicated.”

- many people and companies contributing
- recent new features like prepared statements
- most DBAs probably know how to use it, or can learn it in a day
- drawbacks of single process nature only apply at rather large scales (and then you can still switch to multiple pgbouncers)

pgcat

pgcat is probably the most serious contender

- there are different people and companies contributing to it
- they seem to steadily move forward with features
- Rust should enable them to easily add new features without running into memory management problems

pgagroal

trying to prematurely optimize features, rewriting things completely

- missing TLS encryption in server connections
- lack of queueing
- no lifetime limit for server connections
- why does it use a process model?

odyssey

odyssey started a new era of experimentation with poolers, and kicked pgbouncer development into a higher gear

- odyssey is primarily developed by and for Yandex, which in itself can be an ethical problem
- there are also real problems like memory leaks that are still waiting for a fix
- and then there are issues that I can’t even understand, because some tickets are only written and updated in Russian
- there are no packages

Closing Thoughts

Closing Thoughts

- there are cases where a pooler is simply not necessary -> KISS
- just try out a pooler if you think you need one
  - do lots and lots of testing, especially for correctness
- some problems can be solved with session pooling
- some problems can only be solved with transaction pooling
  - understand if your application relies on things that cross transaction boundaries
  - do more testing

Bonus Slides: Architecture

Where to Place Pooler?

- on application host
- on database host
- on seperate host
- on database host *and* on application host

Pooler on Application Host

- doesn’t solve problem with many application hosts
- don’t need high availability in pooler, just switch to different application host
- PostgreSQL will know about source of connections
- application to pooler doesn’t need TLS
- application to pooler auth can potentially be simpler
- connection creation to pooler is fast
- idle connections are not affected by DB failover
- pooler eats CPU cycles from application host

Pooler on Database Host

- can’t use read-only features (unless another hop is acceptable)
- switch to different DB host when pooler is unhealthy
- everything looks like local connections to PostgreSQL
- pooler to PostgreSQL doesn’t need TLS
- pooler to PostgreSQL auth can potentially be simpler
- connection creation to pooler is slower
- all connections need to be re-established when a primary failover happens
- pooler eats CPU cycles from database host

Pooler on Separate Host(s)

- solves cases where there are too many application hosts and read only routing is necessary
- need HA mechanism (library support, virtual IP, HAProxy . . .)
- can add more poolers as needed
- TLS needed on both sides
- additional hops increase latency in general
- auth is more complicated (pg_hba in PostgreSQL, auth config in pooler)

Pooler on Database Host *and* Pooler on Application Host

can also be different poolers, one *in* the application (e.g. HikariCP) and one on the database host

- combine the advantages
  - client to pooler needs no TLS or complicated auth
  - pooler to server needs no TLS or complicated auth
  - connectiion creation from client to pooler is cheap
  - you can just keep a large enough pool of connections between poolers
  - can reasonably limit number of connections to server even with multiple poolers on application hosts
- combine the disadvantages
  - more things to configure, maintain, troubleshoot
  - pooler eats CPU cycles from database host *and* application host