Manan Gupta

Founding Engineer - Multigres

@GuptaManan100

supabase



Connection Pooling Beyond PgBouncer

A custom approach for distributed PostgreSQL

supabase

Manan Gupta // manan@supabase.io

March 2026

What is Multigres?

Multigres is Vitess for PostgreSQL.



Open Source



Sharding



High Availability



Backups And Restore

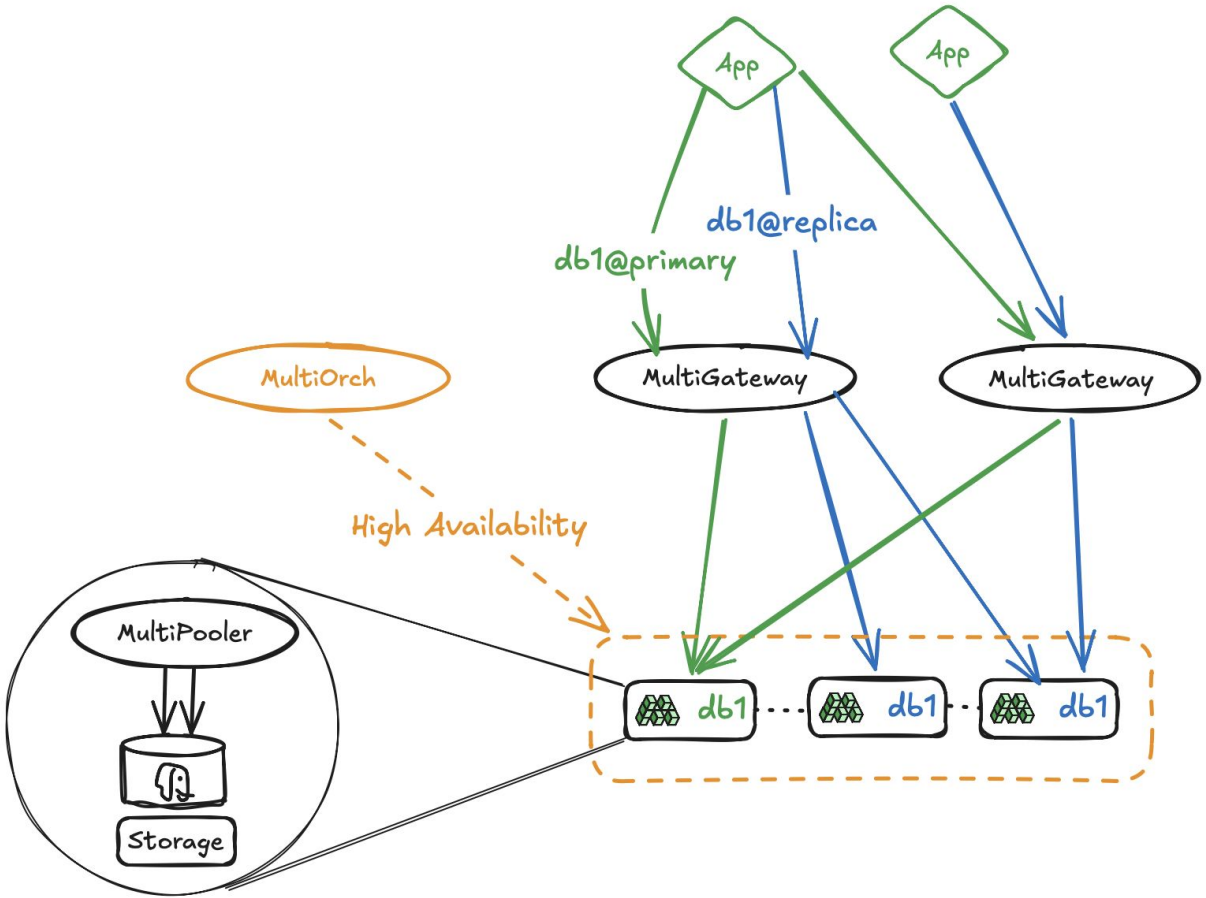


Connection Pooling



Query Analytics

The Architecture



Postgres already has a PgBouncer.
Why build another one?

Sharded Postgres needs a split

Two requirements drive the two services.

1

A single client connection has to reach any shard.

Routing has to live somewhere. That somewhere can't be inside a single postgres.

2

Cross-shard aggregation belongs above any individual postgres.

Combining results from multiple shards is its own job, not the connection pool's.

Multigateway handles both. **Multipooler** manages one postgres's connections.

Cluster Integration

A multigres cluster is a coordinated thing.

Failover

Roles flip. Drain in-flight requests before promoting.

Backups

Coordinate with traffic. The pooler has to know.

Topology

Instances come and go. New shards spin up.

Drain

Close existing txns cleanly, refuse new ones.

PgBouncer is a Sidecar. **Multipooler is part of the cluster.**

Three features in Multigres

...that PgBouncer doesn't have.

01 **Per-user fair pools**

Privilege isolation + max-min fairness.

02 **Pooling without picking a mode**

Transaction-mode speed, session-mode safety - automatic.

03 **Cross-gateway prepared statement consolidation**

One name on postgres for the same query - no matter which gateway forwards it.

Demo lives in feature 02.

Why per-user pools?

Shared pool + SET ROLE

Risk: privilege escalation.

Connect as one privileged user.
SET ROLE to whoever made the request.

Per-user pools (multipooler)

Privilege escalation, by design, can't happen.

Every user gets their own pool.
Each connection is authenticated as that user.

No impersonation. No privileged role to impersonate from.

Many pools.
Fixed connection budget.

One noisy user could starve everyone else.

Max-Min Fairness

Maximize the minimum allocation any user gets. Pool size 12.

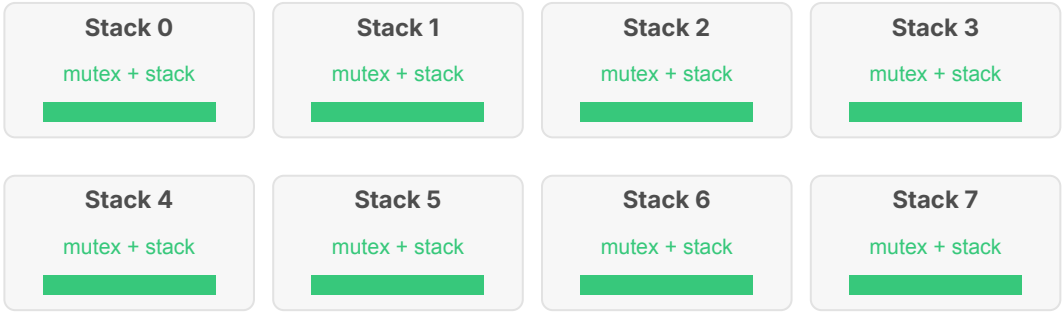
Round	Alice (d=2)	Bob (d=5)	Charlie (d=10)	Used
1	2 ✓	2	2	6 / 12
2	—	5 ✓	5	12 / 12
3	—	—	capped	—
Final	**2**	**5**	**5**	**12 / 12**

- **Noisy neighbors contained**
  Fixed budget split, no monopoly.
- **Pareto optimal**
  Can't give one user more without taking from another.
- **Strategy-proof**
  Claiming 10× your real demand doesn't help.

Eight-stack bucketing

Same user, different settings ≠ same connection.

1. search_path = analytics
   bucket 17 → **stack 1**
2. search_path = reporting
   bucket 22 → **stack 6**
3. timezone = UTC
   bucket 9 → **stack 1**



bucket & 7 → stack number

Feature Summary

Privilege isolation

No SET ROLE.
Every connection is the user it serves.

Fair sharing

Max-min fairness.
No noisy neighbors. Strategy-proof.

Smart reuse

8 stacks by settings.
Hot settings find ready connections.

The mode tradeoff

	PgBouncer Statement	PgBouncer Transaction	PgBouncer Session	Multigres
Multi-statement transactions	✗	✓	✓	✓
Temp tables across queries	✗	✗	✓	✓
LISTEN NOTIFY	✗	✗	✓	✓
Suspended portals / cursors	✗	✗	✓	✓
Connection reuse	High	High	Low	**High***

** high when no reservation reasons are set; moves to reserved only when needed.*

We don’t make you pick

1 Parse every query *fully*

We know exactly what each statement is doing - BEGIN, CREATE TEMP TABLE, COPY, anything.

2 Track session state *explicitly*

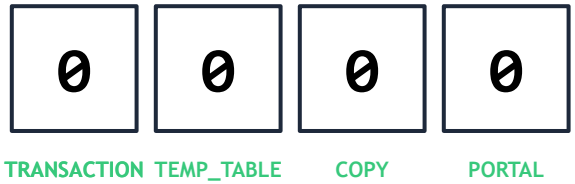
A bitmask per connection. Each bit tracks one reason the connection has to stay reserved.

The pool knows exactly when it's safe to recycle a connection, and when it isn't.

The four reservation reasons

`TRANSACTION`	Set on BEGIN. Cleared on COMMIT or ROLLBACK.
`TEMP_TABLE`	Set when a temp table is created. Cleared on DISCARD TEMP or disconnect.
`COPY`	Set when COPY IN/OUT starts. Cleared when streaming completes.
`PORTAL`	Set when a named portal is suspended. Cleared on full execute or close.

A bitmask per connection



All bits clear → regular pool.

Any bit set → reserved.

Composable - Multiple reasons can hold at once.

One client's session

`BEGIN`	0 0 0 **1**	Reserved	*Transaction starts.*
`CREATE TEMP TABLE t (...)`	0 0 **1** **1**	Reserved	*Two reasons now.*
`INSERT INTO t VALUES (...)`	0 0 **1** **1**	Reserved	*No change.*
`COMMIT`	0 0 **1** 0	Reserved	***TRANSACTION clears. TEMP_TABLE remains.***
`DISCARD TEMP`	0 0 0 0	**Regular**	*All clear. Back to pool.*

Step 4: PgBouncer transaction mode would silently drop the temp table here. Multigres doesn't.

Live Demo!

Multigres vs PgBouncer

Same SQL. Different pool. Different outcome.

Three pool tiers

Regular

no reasons set

Recyclable. Most queries served from here.

Reserved

≥ 1 reason set

Pinned to one client. Released when all clear.

Admin

always available

Small, separate. Management Ops.

What if a client disconnects mid-transaction?

Inactivity killer

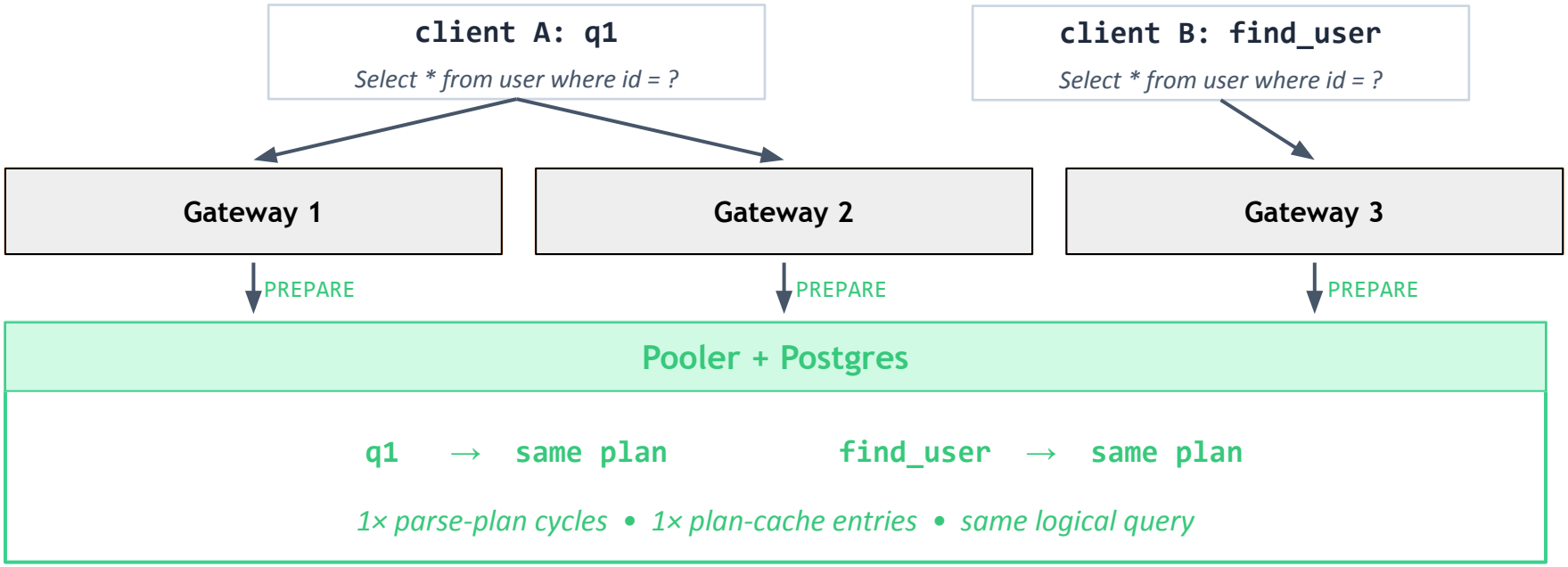
If a reserved connection sits idle past a tunable timeout, the pool kills it with

`pg_terminate_backend`

and frees the budget. The client's session is gone - the budget is recovered.

Reserved connections can't leak forever.

The N-gateways and clients problem



Multiply by N gateways and the client reconnect rate.

How we dedupe

01 **Gateway parses every PREPARE.**

Extracts the query text and parameter types. Forwards those - not the client-chosen name - to multipooler.

```
PREPARE my_query AS …   →   (query_text, [int])
```

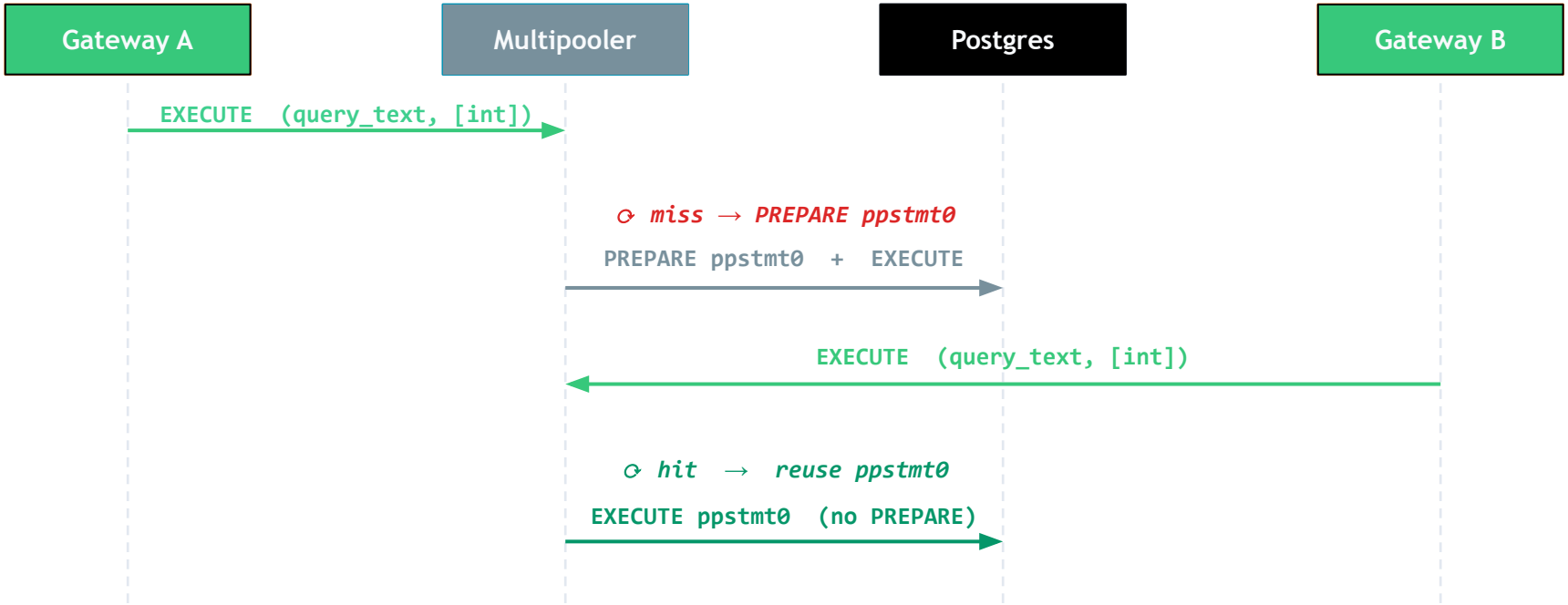
02 **Pooler maps (query_text, param_types) → canonical_name.**

Same query text and types from any gateway → same lookup → same canonical name on the backend.

```
(query_text, [int])   →   ppstmt0
```

The dedup key is the query, not the name. That's what parsing buys us.

The handoff



First time on a backend connection: PREPARE + EXECUTE. After that: EXECUTE only. Canonical name is stable across the whole pooler.

Feature Summary

Same canonical name on every backend connection.

No matter which gateway forwards the query.

Because we parse the query, we dedupe by **query text** - not by client-chosen name.

Tradeoffs we accepted

- **Two services to operate.**

  Not one. Deploy, monitor, upgrade as a coordinated unit.

- **Extra hop on the hot path.**

  Gateway → pooler over gRPC. Small (colocated), but real.

- **More work in the pool.**

  Bitmask, parsing, prepared-statement maps.

- **Detection must be exhaustive.**

  Missing a reason means a wrong recycle.

Worth it for what we get.

Three features. One pooler.

01

Per-user fair pools

Privilege isolation. Max-min fair sharing across users.

02

Pooling without picking a mode

Transaction-mode speed, session-mode safety - automatic.

03

Cross-gateway prepared statement consolidation

One name on postgres for the same query - no matter which gateway forwards it.

Read more

4 PART BLOG SERIES

1. Two jobs, two processes: why multigres has its own connection pooler
2. Per-user pools that share fairly
3. Pooling without choosing a mode
4. One parse per query, no matter how many gateways

Link

`multigres.com/blog/two-jobs-two-processes`



Thank you

Questions?

manan@supabase.com

Manan Gupta • Multigres

github.com/multigres/multigres • multigres.com

Get the slides

Enter to win a new Mac Mini

supabase.link/pgconfdev-2026

Slides from this talk
Get the deck and notes.

Enter to win
Signup for Supabase, load data, and enter the contest.

Contact me
manan@supabase.com
@GuptaManan100



github.com/multigres/multigres • multigres.com

supabase