PgBouncer at scale
Multi-instance setup

data egret

Your remote PostgreSQL DBA team

Stefan FERCOT

stefan.fercot@dataegret.com

PGDay Lowlands
September 2025

Securing your PostgreSQL database availability and high performance.

- Performance audit
- Backup & restore
- Migration
- Cloud Cost Management
- Architecture review
- DataOps/ CDC projects
- 24/7 Incident support

on premises & cloud

EXPERTISE

Senior DBA team with **10+ years of PostgreSQL experience** each.

DEVELOPMENT

Involved in **new feature and extension development**.

TAILORED APPROACH

Dedicated DBA team that focused on success of your project.

COMMUNITY

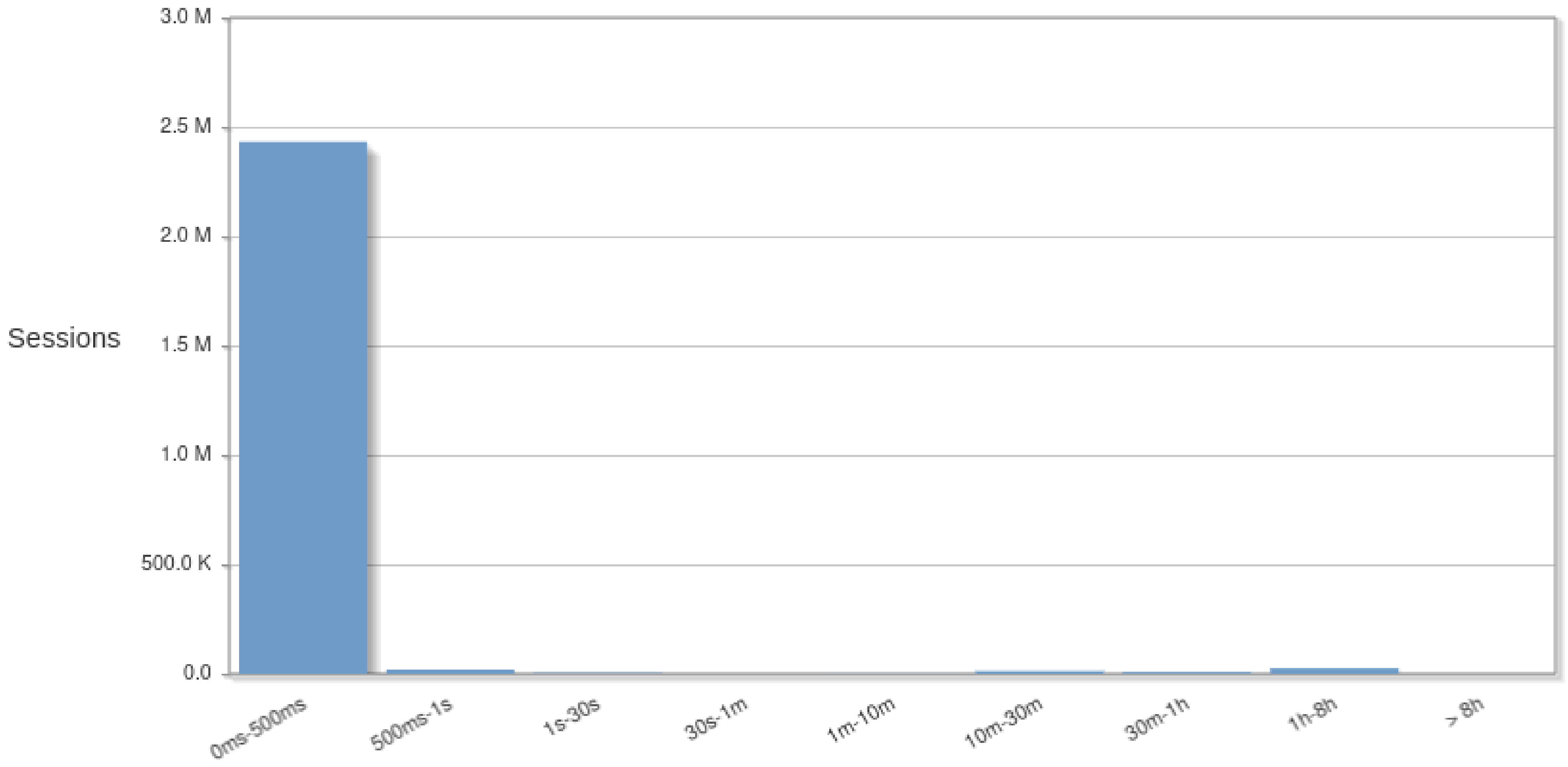
Recognised significant **contributing sponsor to PostgreSQL**.

Stefan Fercot

- PostgreSQL Expert **@Data Egret**
- pgBackRest contributor
- aka. pgstef
- https://pgstef.github.io

PgBouncer at scale
Multi-instance setup

Rings a bell?


Sessions
3.0 M
2.5 M
2.0 M
1.5 M
1.0 M
500.0 K
0.0
0ms-500ms
500ms-1s
1s-30s
30s-1m
1m-10m
10m-30m
30m-1h
1h-8h
> 8h

Kudos

Peter Eisentraut

Today’s agenda

- Why PgBouncer?
- Running multiple instances
- Query cancellation
- Example config
- Lessons learned

Why use PgBouncer?

- Connections are expensive -> pooling helps
- Centralized connections control
- Protects clients from disconnections:
  - Restarts (upgrades / failovers)
  - Connection spikes or saturation

Drawbacks of connection pooling

- Feature limitations depending on the pooling mode
- Potential performance issues if misconfigured:
  - Latency overhead or throughput bottlenecks
- Authentication handling can be tricky
- Adds complexity to the architecture
- Single Point of Failure (SPOF) if not redundant

Why scale PgBouncer?

```
Clients ---> [PgBouncer] ---> PostgreSQL
               (1 core)
```

- PgBouncer is a **single-threaded process** -> one CPU only
- One instance can handle ~10k connections (≃1k active)
- Limits vary with config and amount of data fetched
- For higher loads: adjust system limits or run multiple instances!

Running more than one PgBouncer

```
Clients ---> [PgBouncer #1] --\
        ---> [PgBouncer #2] ---+--> PostgreSQL
        ---> [PgBouncer #3] --/
         (kernel load balance)
```

- Run multiple PgBouncer instances in parallel
- Each listens on the same port, kernel distributes traffic
- Requires Linux `SO_REUSEPORT` socket option

The query cancellation problem

```
Cancel request
    |
    v
 [PgBouncer #1]  (doesn't own session)  ---> fails
 [PgBouncer #2]  (owns session)         ---> should cancel
```

- PostgreSQL supports **query cancellation** (`pg_cancel_backend()`)
  - Via `CancelRequest` message sent in separated connection
- With multiple PgBouncers, requests may hit the wrong instance
- If that instance doesn’t own the connection -> **cancellation fails**
- https://dataegret.com/2024/08/handling_cancellation_request/

PgBouncer peering

```
Cancel request
    |
    v
 [PgBouncer #1] --forwards--> [PgBouncer #2] ---> PostgreSQL
```

- **Solution:** use the PgBouncer peering feature
- Configure `[peers]` section + `peer_id` for each instance
- Each PgBouncer knows its siblings and forwards cancellations
- Ensures cancel requests reach the correct process

Example setup

- Use `ReusePort=true` in **systemd** socket unit
- Set `peer_id` in each **PgBouncer** config
- All instances share port **6432**
- PostgreSQL server: `host=postgres0 port=5432`
- Unix sockets:
  - `/run/postgresql/.s.PGSQL.10001`,
  - `/run/postgresql/.s.PGSQL.10002`

PgBouncer peer 1 configuration

`/etc/pgbouncer/pgbouncer-10001.ini`

```
[databases]
testdb = host=postgres0 port=5432 dbname=testdb

[peers]
1 = host=/run/postgresql port=10001
2 = host=/run/postgresql port=10002

[pgbouncer]
listen_addr = 0.0.0.0
listen_port = 6432
peer_id = 1
```

PgBouncer peer 2 configuration

`/etc/pgbouncer/pgbouncer-10002.ini`

```
[databases]
testdb = host=postgres0 port=5432 dbname=testdb

[peers]
1 = host=/run/postgresql port=10001
2 = host=/run/postgresql port=10002

[pgbouncer]
listen_addr = 0.0.0.0
listen_port = 6432
peer_id = 2
```

systemd service unit template

`/etc/systemd/system/pgbouncer@.service`

```
[Unit]
Description=connection pooler for PostgreSQL (%i)
After=network.target
Requires=pgbouncer@%i.socket

[Service]
Type=notify
User=postgres
ExecStart=/bin/pgbouncer /etc/pgbouncer/pgbouncer-%i.ini
ExecReload=/bin/kill -HUP $MAINPID
KillSignal=SIGINT

[Install]
WantedBy=multi-user.target
```

systemd socket unit

`/etc/systemd/system/pgbouncer@.socket`

```
[Unit]
Description=sockets (%i) for PgBouncer

[Socket]
ListenStream=0.0.0.0:6432
ListenStream=0.0.0.0:%i
ListenStream=/run/postgresql/.s.PGSQL.%i
ReusePort=true

[Install]
WantedBy=sockets.target
```

Activate the PgBouncer sockets

```
$ sudo systemctl enable --now pgbouncer@10001.socket
$ sudo systemctl enable --now pgbouncer@10002.socket
```

```
$ sudo systemctl list-sockets | grep pgbouncer
0.0.0.0:10001                       pgbouncer@10001.socket      pgbouncer@10001.service
0.0.0.0:10002                       pgbouncer@10002.socket      pgbouncer@10002.service
0.0.0.0:6432                        pgbouncer@10002.socket      pgbouncer@10002.service
0.0.0.0:6432                        pgbouncer@10001.socket      pgbouncer@10001.service
/run/postgresql/.s.PGSQL.10001      pgbouncer@10001.socket      pgbouncer@10001.service
/run/postgresql/.s.PGSQL.10002      pgbouncer@10002.socket      pgbouncer@10002.service
```

Minimizing downtime

```
Clients --paused--> PgBouncer --queries wait--> PostgreSQL
```

- How to route traffic and avoid downtime during the maintenance task?
  - No need to drop client traffic
  - Use PgBouncer `PAUSE` and `RESUME`
- Useful for:
  - Failovers
  - Rolling upgrades
  - Tips and tools for minimal downtime in PostgreSQL upgrades

```
$ psql -U pgbouncer -p 10001 -c "PAUSE;"
$ psql -U pgbouncer -p 10001 -c "RESUME;"
```

Lessons learned

```
Scale-up path:
  2 instances -> 4 -> 8
  Watch CPU & connections per core
```

- Start small and scale gradually
- Monitor connection distribution (kernel load balancing is not always even)
- More processes = more configs + logs to manage
- Keep PgBouncer and PostgreSQL logs separated
- Tune `ulimit` and system-level connection limits
- Use multiple instances + `peer_id` to avoid SPOF

Conclusion

- PgBouncer = most mature and widely used PostgreSQL pooler
  - Homepage: https://www.pgbouncer.org/
  - Sources: https://github.com/pgbouncer/pgbouncer
- Scales beyond single-core with multi-instance setup
  - **SO_REUSEPORT + systemd** make it simple
  - **Peering** fixes cancellation issues
  - **PAUSE/RESUME** minimizes downtime

Final thoughts

> *Don’t be scared, be prepared*

- PgBouncer scales gracefully under real-world load
  - Helps overcome some PostgreSQL limitations
- Using proper tools helps manage downtime
  - Both when things go right and when they don’t

PostgreSQL 💙 Belgium

- PgBE PostgreSQL Users Group Belgium meetup group
  - **October 14** – Google, Brussels
  - **November 25** – Idewe, Leuven

Thank you!

contact@dataegret.com