



Per-Backend Statistics

See What Every Connection Is Actually Doing

Bertrand Drouvot
Principal Software Engineer
Amazon Web Services
PGConf.EU 2026

About me

- Working in databases for about 25 years
- Oak table member
- Oracle ACE



About me

- Working in databases for about 25 years
- Oak table member
- ~~Oracle ACE~~
- Fell in love for PostgreSQL
- Working at AWS (RDS PostgreSQL contributor team)
- Major contributor
- Twitter: @BertrandDrouvot
- <https://bdrouvot.github.io/>



Agenda

What is it about?

There's a spike in I/O

`pg_stat_get_backend_io()` to the rescue

pgstats architecture overview

Current design and its limitations

New design

What's landed and what's next

Live demo



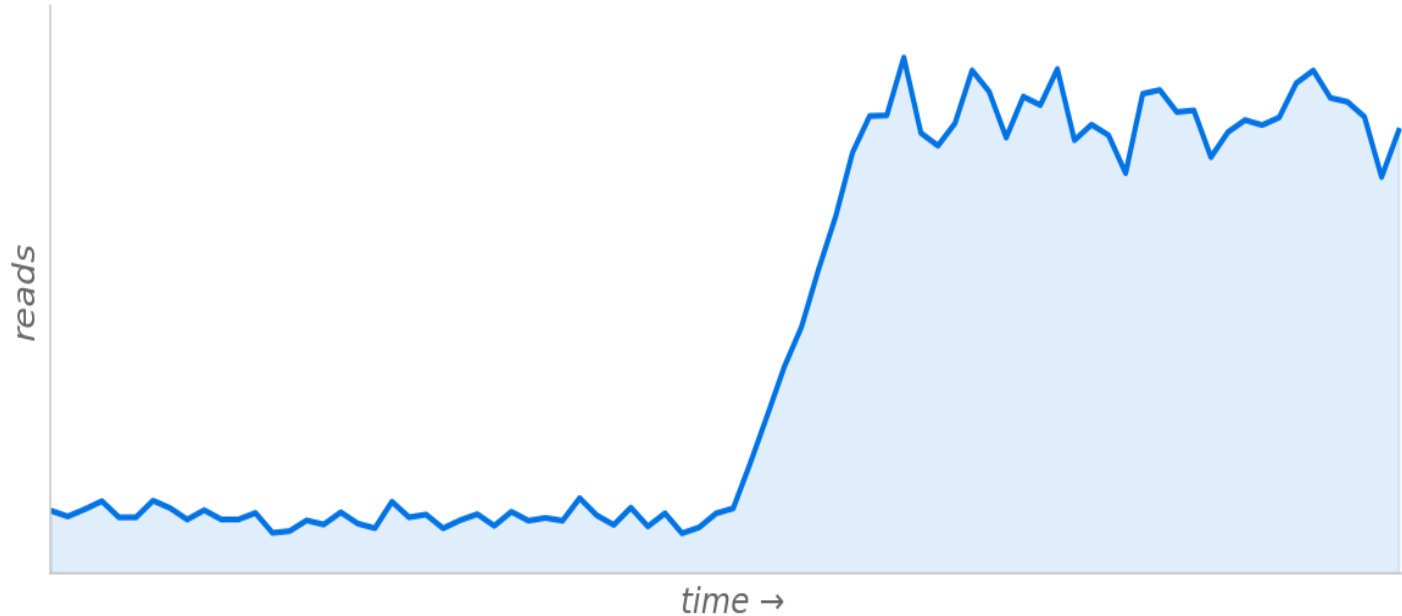
What is it about?

Monitoring and Observability

PostgreSQL 18 adds more details to the `EXPLAIN` utility, which provides information about query plan execution, and as of this release now automatically shows how many buffers (the fundamental unit of data storage) are accessed when executing `EXPLAIN ANALYZE`. Additionally, `EXPLAIN ANALYZE` now shows how many index lookups occur during an index scan, and `EXPLAIN ANALYZE VERBOSE` includes CPU, WAL, and average read statistics. This release also includes information about the total amount of time spent vacuuming and analyzing a table in `pg_stat_all_tables`, and now shows per-connection statistics on I/O and WAL utilization.

PostgreSQL 18 also provides more insights into write conflicts that occur during logical replication, and surfaces this information both in logs and in the `pg_stat_subscription_stats` view.

There's a spike in I/O.



200 active connections.

Is it one backend doing a seq scan?

Or is everyone reading almost equally?

Which backend(s)?

Ok, let's find who's doing all that I/O...

① pg_stat_activity

Shows: query, state,
wait_event

But: no I/O counters.

You see what's running,
not how much I/O it does.

② OS tools (iotop, pidstat)

Would show per-PID I/O

Requires OS access.

③ pg_stat_io

Shows: reads, writes, hits
Per backend_type, context,
object

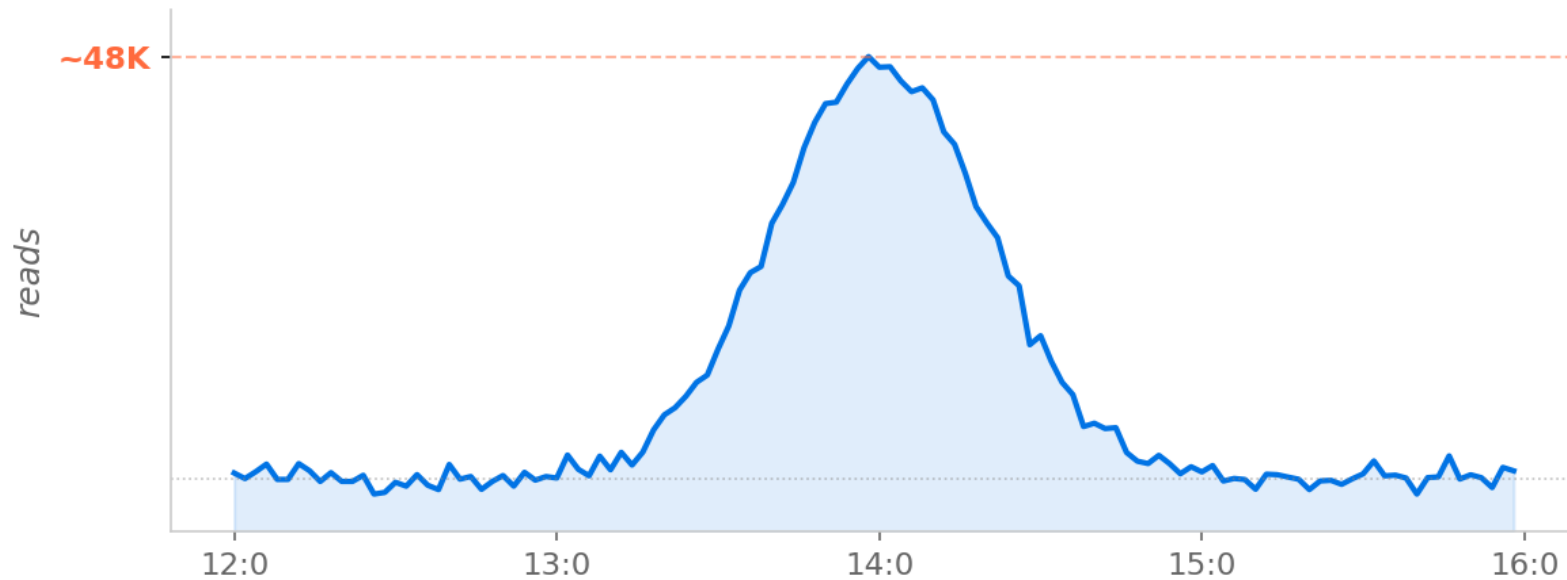
But: not per backend.

All "client backend" reads
are lumped together.

We're stuck.

Any other ideas?

If the spike occurred in the past



① pg_stat_activity

Shows: query, state,
wait_event

But: no I/O counters.

You see what's running,
not how much I/O it does.

② OS tools (iotop, pidstat)

Would show per-PID I/O

Requires OS access.

③ pg_stat_io

Shows: reads, writes, hits
Per backend_type, context,
object

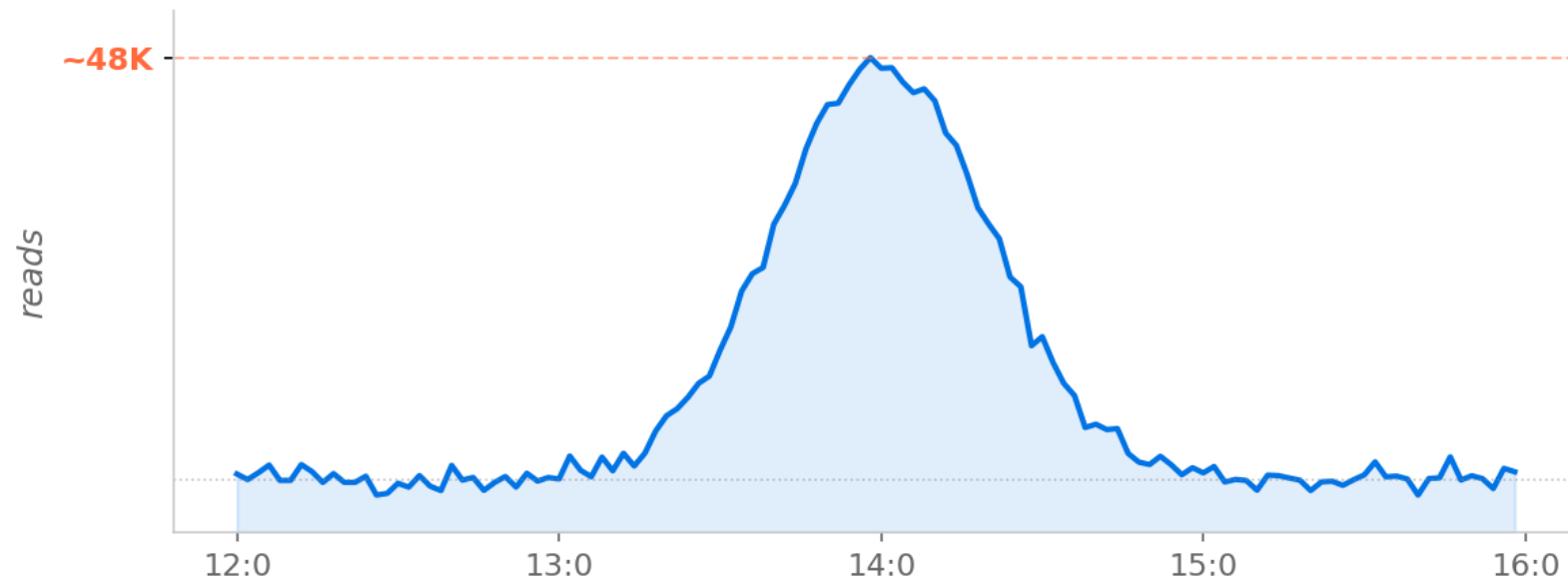
But: not per backend.

All "client backend" reads
are lumped together.

pg_stat_get_backend_io() to the rescue

```
SELECT pid, backend_start, application_name, sum(reads)
FROM pg_stat_activity, pg_stat_get_backend_io(pid)
WHERE reads != 0 GROUP BY 1,2,3 ORDER BY 4 DESC;
```

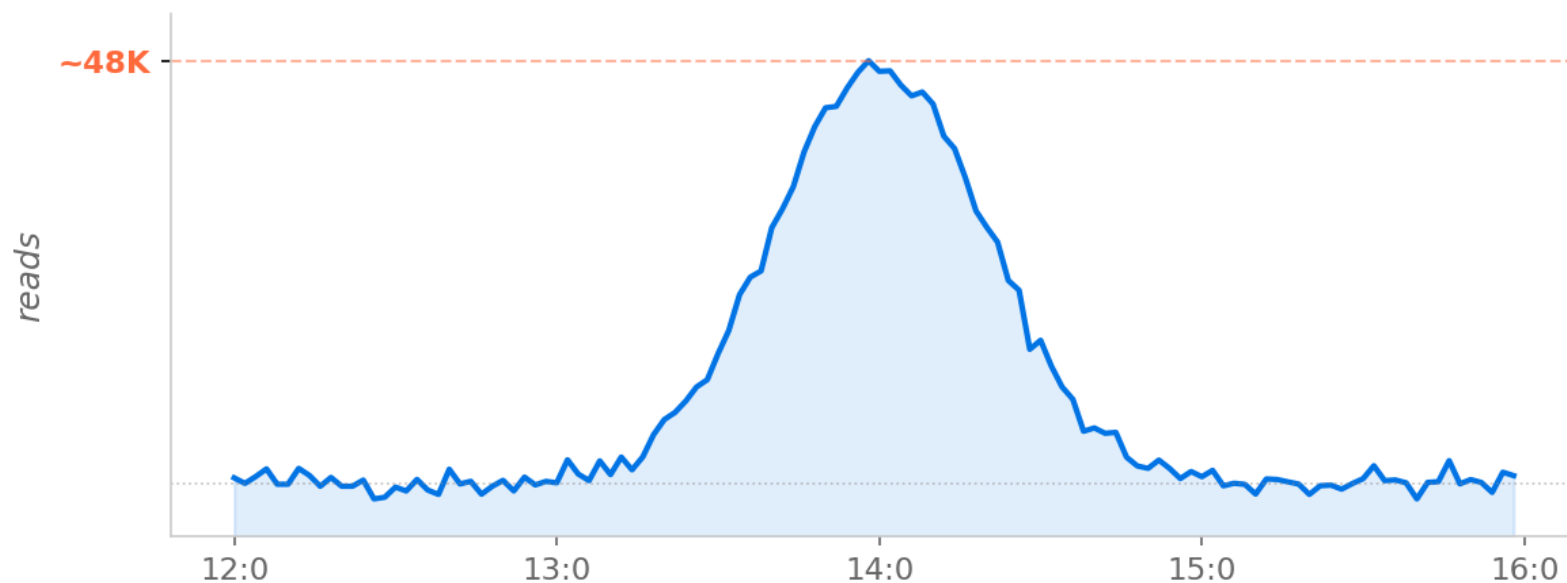
pid	backend_start	application_name	reads
502479	2026-10-02 13:47:12	batch_report	48231
501832	2026-10-02 09:15:42	webapp	127
501890	2026-10-02 09:15:43	webapp	98
502201	2026-10-02 11:30:05	analytics	41
502305	2026-10-02 12:02:18	webapp	12



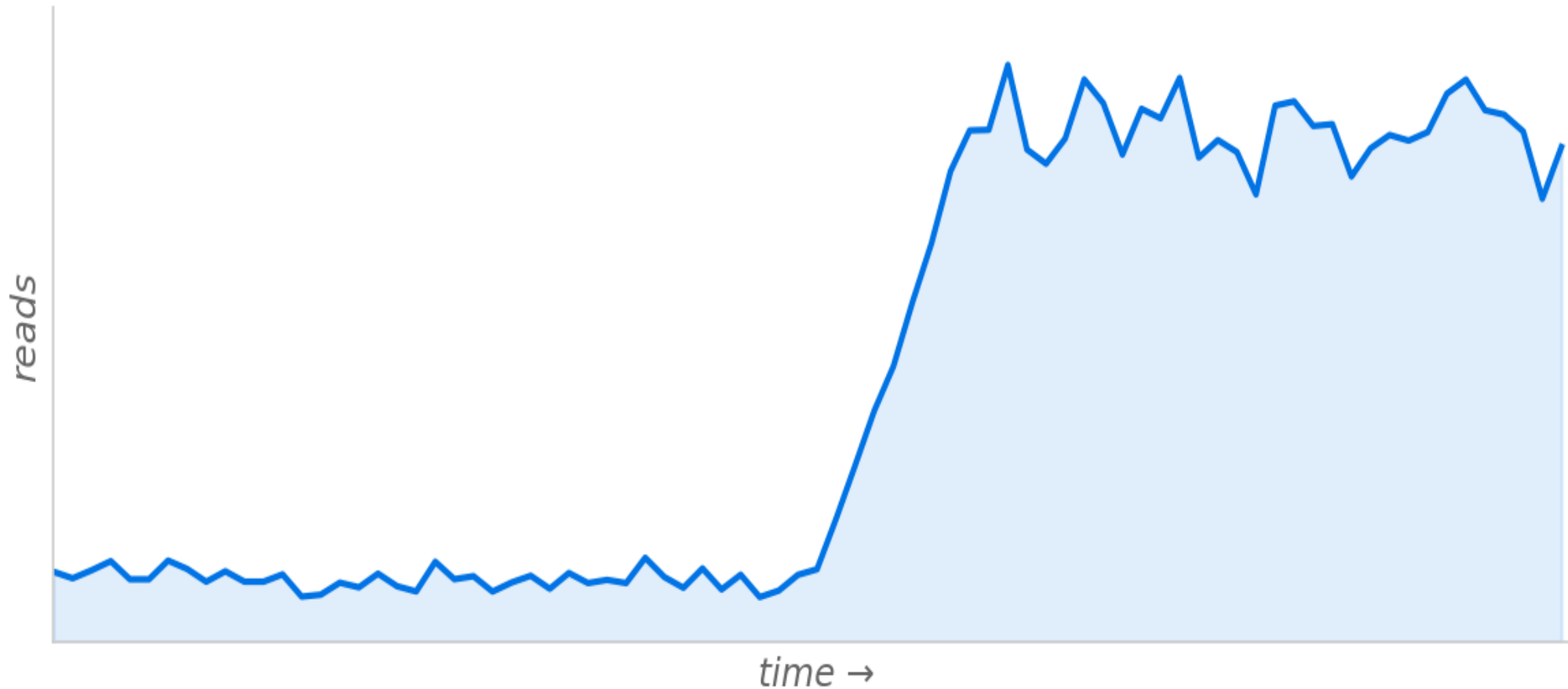
It's not one backend, it's a single application

```
SELECT application_name, sum(reads) AS reads,  
       round(100 * sum(reads) / sum(sum(reads)) OVER(),  
2) AS "%"  
FROM pg_stat_activity, pg_stat_get_backend_io(pid)  
WHERE reads != 0  
GROUP BY application_name ORDER BY 2 DESC;
```

application_name	reads	%
batch_report	48231	99.43
webapp	237	0.49
analytics	41	0.08

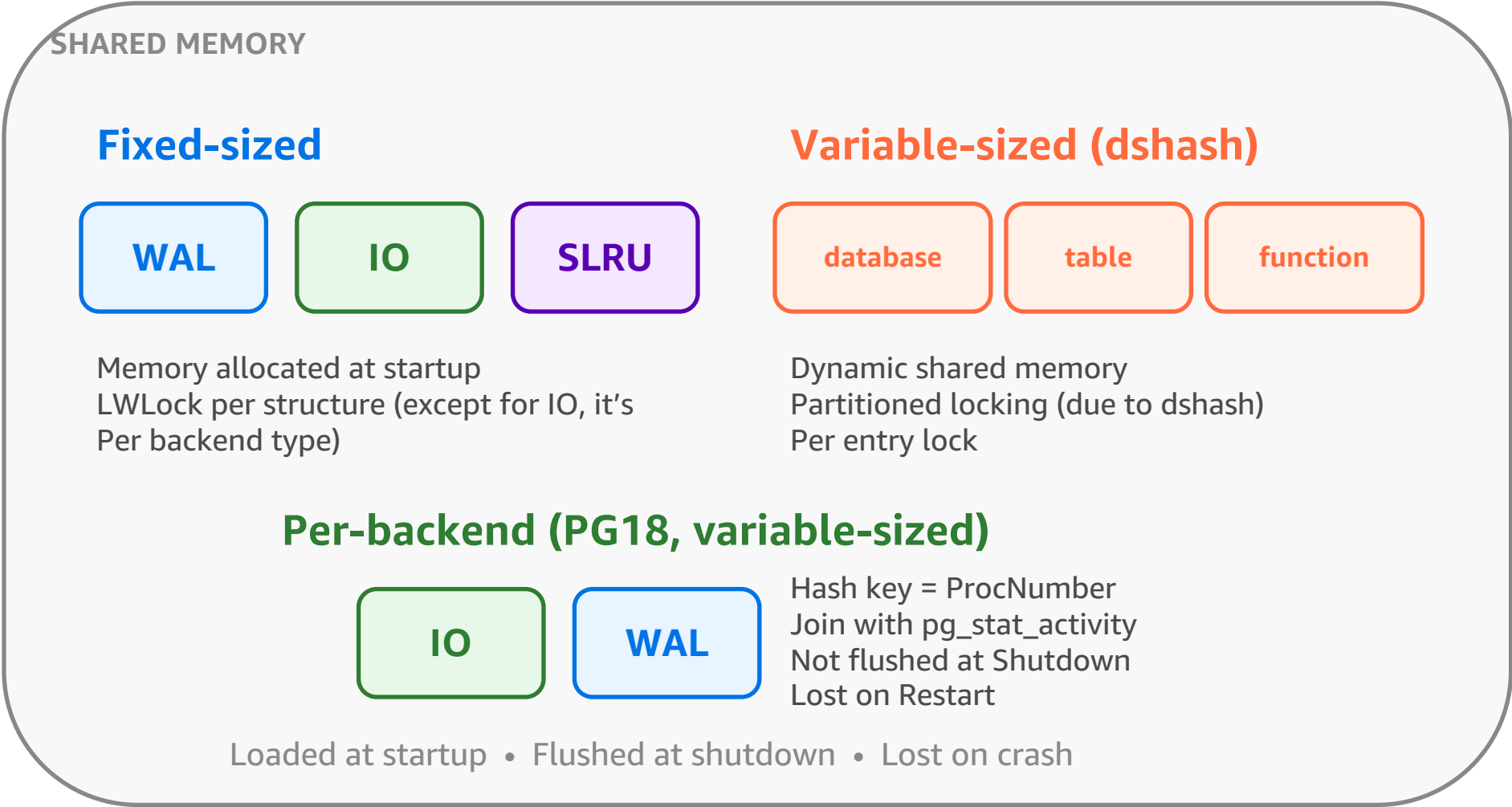


pg_stat_get_backend_io() also obviously helps for what's happening now



pgstats architecture

Each backend reports into shared memory



How stats are reported

LOCAL

Pending stats (per-process)

Flush triggers:

- Transaction ends

Flush (non blocking)

Limits shared memory interaction

SHARED

CLIENT



How stats are reported

LOCAL

Pending stats (per-process)

Flush triggers:

- Transaction ends

Flush (non blocking)

Limits shared memory interaction

✗ lock held → conditional acquire fails

SHARED

Non-blocking attempt : if the entry lock is held, give up immediately.

CLIENT

How stats are reported

LOCAL

Pending stats (per-process)

Flush triggers:

- Transaction ends

Limits shared memory interaction

flush skipped

SHARED

CLIENT



How stats are reported

LOCAL

Pending stats (per-process)



Flush (blocking)

SHARED

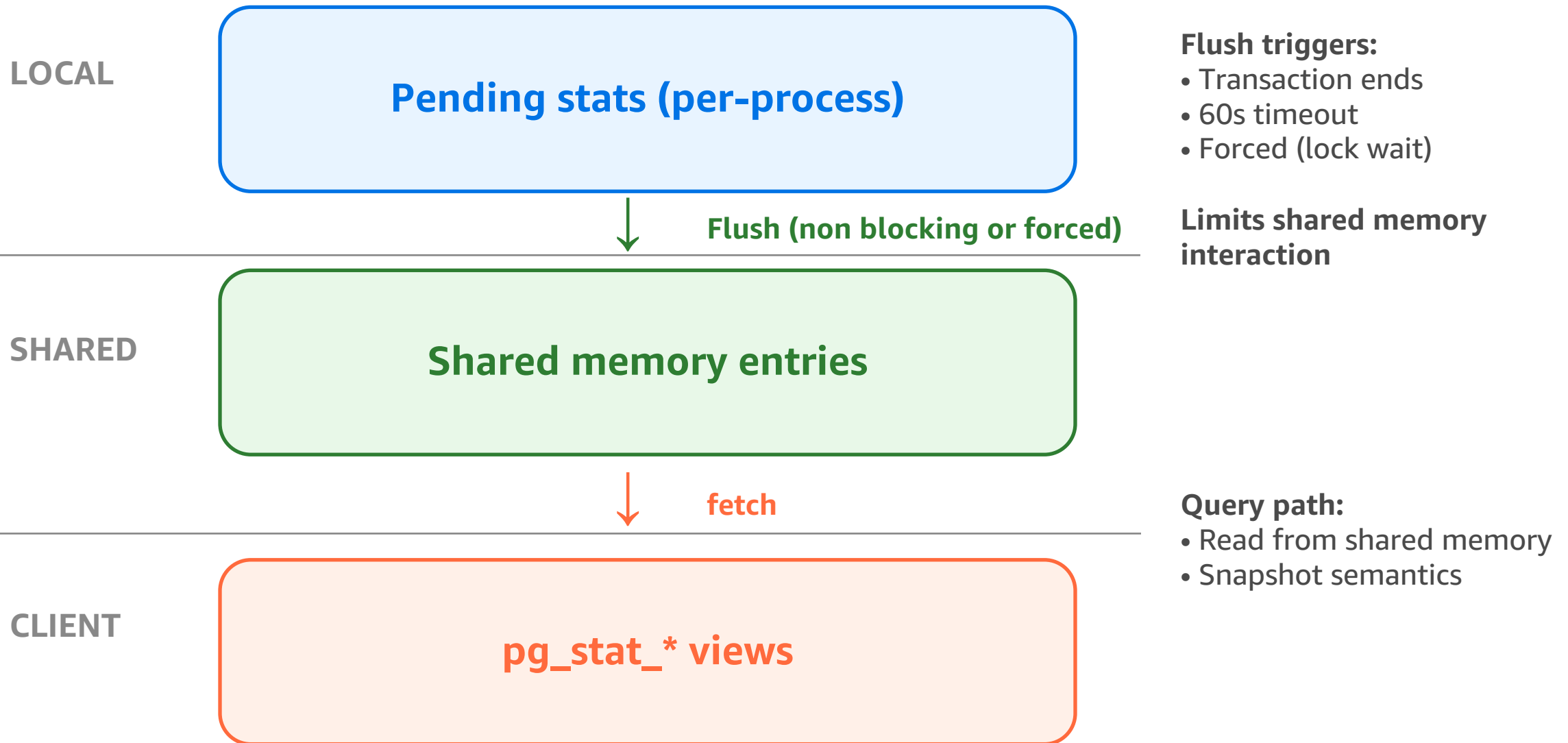
CLIENT

Flush triggers:

- Transaction ends
- 60s timeout, we force flush

Limits shared memory interaction

How stats are reported



Per-backend current design (IO stat example)

LOCAL

PendingBackendStats.pending_io

Per-backend path only
(global view later)

SHARED

PgStatShared_Backend.stats.io_stats

CLIENT

pg_stat_get_backend_io()



IO read → local accumulation

LOCAL

PendingBackendStats.pending_io
READ_bytes: 1024

IO operation increments
local pending counter

SHARED

PgStatShared_Backend.stats.io_stats
READ_bytes: 901120

CLIENT

pg_stat_get_backend_io()



pgstat_count_backend_io_op(), accumulate locally

LOCAL

PendingBackendStats.pending_io
READ_bytes: $1024 + 2048 = 3072$

Direct accumulation.
No current – prev diff needed
(unlike WAL).

SHARED

PgStatShared_Backend.stats.io_stats
READ_bytes: 901120

CLIENT

pg_stat_get_backend_io()



pgstat_count_backend_io_op(), more accumulation

LOCAL

PendingBackendStats.pending_io
READ_bytes: $3072 + 1024 = 4096$

Keeps accumulating
until flush

SHARED

PgStatShared_Backend.stats.io_stats
READ_bytes: 901120

CLIENT

pg_stat_get_backend_io()



pgstat_flush_backend(), transfer to shared

LOCAL

PendingBackendStats.pending_io
READ_bytes: 4096

+= 4096 to shared entry



SHARED

PgStatShared_Backend.stats.io_stats
READ_bytes: 901120

CLIENT

pg_stat_get_backend_io()



Flush complete

LOCAL

PendingBackendStats.pending_io
READ_bytes: 0 (reset)

Local counter reset.
Shared entry updated.

SHARED

PgStatShared_Backend.stats.io_stats
READ_bytes: 905216

CLIENT

pg_stat_get_backend_io()



pgstat_fetch_stat_backend(), client retrieves

LOCAL

PendingBackendStats.pending_io

Direct fetch from
shared entry by ProcNumber

SHARED

PgStatShared_Backend.stats.io_stats
READ_bytes: 905216

CLIENT

pg_stat_get_backend_io()
READ_bytes: 905216



Current design: 3 layers (global view)

LOCAL

PendingBackendStats.pending_io

PendingIOStats

Two local counters.
Two shared
structures.

SHARED

PgStatShared_Backend.stats.io_stats

pgStatLocal.shmem->io.stats

CLIENT

pg_stat_get_backend_io() | pg_stat_io



IO read → double increment

LOCAL

PendingBackendStats.pending_io
READ_bytes: 1024

PendingIOStats
READ_bytes: 1024

pgstat_count_backend_io_op()
+ pgstat_count_io_op()

SHARED

PgStatShared_Backend.stats.io_stats

pgStatLocal.shmem->io.stats

CLIENT

pg_stat_get_backend_io() | pg_stat_io



Double flush

LOCAL

PendingBackendStats.pending_io
READ_bytes: 4096

PendingIOStats
READ_bytes: 4096

pgstat_flush_backend()
+ pgstat_io_flush_cb()

SHARED

PgStatShared_Backend.stats.io_stats
READ_bytes: 901120

pgStatLocal.shmem->io.stats
READ_bytes: 14417920

CLIENT

pg_stat_get_backend_io() | pg_stat_io



Double flush complete

LOCAL

PendingBackendStats.pending_io
READ_bytes: 0

PendingIOStats
READ_bytes: 0

SHARED

PgStatShared_Backend.stats.io_stats
READ_bytes: 905216

pgStatLocal.shmem->io.stats
READ_bytes: 14422016

CLIENT

pg_stat_get_backend_io() | pg_stat_io



Client retrieval (global view)

LOCAL

PendingBackendStats.pending_io

PendingIOStats

SHARED

PgStatShared_Backend.stats.io_stats
READ_bytes: 905216

pgStatLocal.shmem->io.stats
READ_bytes: 14422016

CLIENT

pg_stat_get_backend_io() | pg_stat_io
905216 | 14422016



Current design

Double increment:

```
pgstat_count_backend_io_op() → PendingBackendStats  
pgstat_count_io_op()        → PendingIOStats
```

Double flush:

```
pgstat_flush_backend() → PgStatShared_Backend  
pgstat_io_flush_cb()   → pgStatLocal.shmem->io
```

Same counters, incremented and stored twice.

Consequences:

- Global lock (for the global view) acquired on every flush
- Cost grows with each new per-backend stat type
- WAL, Lock, IO share one PGSTAT_KIND_BACKEND entry + lock

New design: 3 layers

LOCAL

PendingBackendStats (local)

One dshash per kind
(IO, WAL, Lock)

SHARED

dshash entry [ProcNumber] (IO kind)

CLIENT

pg_stat_get_backend_io() | pg_stat_io



IO read → local accumulation (no global update!)

LOCAL

PendingBackendStats (local)
READ_bytes: 4096

Same as before:
accumulate locally first

SHARED

dshash entry [ProcNumber] (IO kind)
READ_bytes: 901120

CLIENT

pg_stat_get_backend_io() | pg_stat_io



pgstat_flush_backend() → single write (no global lock!)

LOCAL

PendingBackendStats (local)
READ_bytes: 4096

Only ONE write.
No global lock.
Cached pointer → no hash lookup.
No hash lookup → no hash lock.



SHARED

dshash entry [ProcNumber] (IO kind)
READ_bytes: 901120

CLIENT

pg_stat_get_backend_io() | pg_stat_io



Flush complete, counter stored once

LOCAL

PendingBackendStats (local)
READ_bytes: 0 (reset)

No duplicate.
Global stats NOT updated here.

SHARED

dshash entry [ProcNumber] (IO kind)
READ_bytes: 905216

CLIENT

pg_stat_get_backend_io() | pg_stat_io



pg_stat_get_backend_io(pid), fetch one entry

LOCAL

PendingBackendStats (local)

Direct lookup by ProcNumber.
Same as before.

SHARED

dshash entry [ProcNumber] (IO kind)
READ_bytes: 905216

CLIENT

pg_stat_get_backend_io(pid)
READ_bytes: 905216



pg_stat_io: aggregate all live entries + exited

LOCAL

PendingBackendStats (local)

Tradeoff?
Aggregation moved to query path
(sub-ms at 400 backends)

SHARED

dshash entries (all live)
 Σ READ_bytes: 1,200,000

exited_accumulator (IO kind)
READ_bytes: 250000



+



CLIENT

pg_stat_io
READ_bytes: 1,450,000



Backend exits, entry transferred to accumulator

LOCAL

PendingBackendStats (local)
(backend gone)

Entry removed from dshash.
Counters preserved in exited_accumulator.
Nothing is lost.

SHARED

dshash entry [ProcNumber] (IO kind)
READ_bytes: 905216



exited_accumulator
READ_bytes: 250000 + 905216

CLIENT

pg_stat_get_backend_io() | pg_stat_io



Backend exits, entry transferred to accumulator

LOCAL

PendingBackendStats (local)
(backend gone)

Entry removed from dshash.
Counters preserved in exited_accumulator.
Nothing is lost.

SHARED

dshash entry [ProcNumber] (IO kind)
READ_bytes: 0

exited_accumulator
READ_bytes: 1,155,216

CLIENT

pg_stat_get_backend_io() | pg_stat_io



pg_stat_io: aggregate all live entries + exited

LOCAL

PendingBackendStats (local)

SHARED

dshash entries (all live)
 Σ READ_bytes: 294,784

exited_accumulator (IO kind)
READ_bytes: 1,155,216

Flushed at shutdown
Loaded at startup



+



CLIENT

pg_stat_io
READ_bytes: 1,450,000



The new design: overview

Each counter stored once.

Nonblocking flush path:

- updates only the process's per-backend entry for that kind
- locks only that entry
- takes no kind-global lock and performs no hash lookup because the entry pointer is cached

A forced (blocking) flush can perform a hash lookup and create a missing entry.

Global query (`pg_stat_io`, `pg_stat_wal`, `pg_stat_lock`):

= `exited_accumulator` + $\Sigma(\text{live entries})$ (extra cost)

Per-kind isolation:

each kind has its own dhash, accumulator lock and per-entry lock

- Adding a new stat type doesn't contend with existing ones

Auxiliary processes now included.

The new design at a glance

Area	New design	Benefit
Global view	Routine flushes update only the process's per-backend entry	Removes the common global-stats lock from the frequent flush path
Global view	Global values = live entries + exited-process accumulator	Counters stored once; no separate global + per-backend updates
Global view	Queries lock entries individually	A global query no longer blocks all reporters via one common lock
Global view	Stats transferred to accumulator at process exit / ProcNumber reuse	Preserves stats after the per-backend entry disappears
Per-backend	Each process and stat kind has its own entry and lock	One process or kind doesn't contend with unrelated ones
Per-backend	Entries created and cached during <code>pgstat_initialize()</code>	No hash lookup or allocation on flush; supports non-InitPostgres workers
Per-backend	Entries allocated only for processes that exist	Avoids reserving storage for every possible ProcNumber
Per-backend	Generic snapshot / caching / accumulation / reset infra	Consistent across WAL, I/O, lock, and future stat kinds



what's landed: Per-backend I/O statistics (PG18)

`pg_stat_get_backend_io(pid)`: the per-backend counterpart of `pg_stat_io`

Column	Type
backend_type	text
object	text
context	text
reads	bigint
read_bytes	numeric
read_time	double precision
writes	bigint
write_bytes	numeric
write_time	double precision
writebacks	bigint
writeback_time	double precision
extends	bigint
extend_bytes	numeric
extend_time	double precision
hits	bigint
evictions	bigint
reuses	bigint
fsyncs	bigint
fsync_time	double precision
stats_reset	timestamp with time zone

This is the function that answered “which backend?” at the start of the talk.



what's landed: Per-backend WAL statistics (PG18)

`pg_stat_get_backend_wal(pid)`: the per-backend counterpart of `pg_stat_wal`

Column	Type
wal_records	bigint
wal_fpi	bigint
wal_bytes	numeric
wal_fpi_bytes	numeric
wal_buffers_full	bigint
stats_reset	timestamp with time zone

what's landed: Per-backend Lock statistics (PG20)

`pg_stat_get_backend_lock(pid)`: the per-backend counterpart of `pg_stat_lock` (PG 19)

Column	Type
locktype	text
waits	bigint
wait_time	double precision
fastpath_exceeded	bigint
stats_reset	timestamp with time zone

what's landed: Per-backend Lock statistics (PG20)

-  **Gülçin Yıldırım Jelínek** @apatheticmagpie · 2 sept.  ...
Wrote about the new system views in PG19. I think this might be the longest I've ever spent on a blog 😊
- I expanded on the `pg_stat_lock` (cumulative, cluster-wide lock statistics with one row per lock type). A lot on `pg_stat_recovery` & `pg_stat_autovacuum_scores`. All with queries!
- 2 3 16 771
-  **Buswe** @buswe_com · 2 sept.  ...
`pg_stat_lock` is the one I'm most looking forward to: cumulative counters beat cron-sampling `pg_locks` and hoping you catch the spike.
- 1 1 21
-  **Gülçin Yıldırım Jelínek** @apatheticmagpie · 2 sept.  ...
Yes, it is super exciting. I go deeper on `fastpath_exceeded` counter in the blog. Super cool stuff.
- 1 21
-  **Bertrand Drouvot** @BertrandDrouvot · 3 sept.  ...
FWIW, 20 should have the per-backend side of it: git.postgresql.org/gitweb/?p=post... (useful to see which application/backend is "suffering" if any).
- 1 1 21
-  **Gülçin Yıldırım Jelínek** @apatheticmagpie  ...

 [Afficher la traduction](#)

Nice, even better! Thanks for the heads-up Bertrand.

what's next: Per-backend AIO statistics (PG20?)

- **Proposed asynchronous I/O accounting per backend**

More about « how the I/O was done » than « what it did ».

Would help for:

- Fallback to sync? (TOAST chunks, temp buffers...)
- Completed by self or another process? (io_uring)
- Handling waits? (io_max_concurrency too low?)

Not yet committed, design in progress, alongside the per-backend redesign

what's next: Per-backend statistics redesign (PG20?)

Separately, I'm doubtful it makes sense to have only per-backend stats for this. I think you'd almost always want the stats for exited backend (e.g. parallel workers) too.

Unfortunately I'm pretty doubtful that `pgstat_backend.c` is the right architectural direction. It'll just end up implementing all kinds of stats, since we'll incrementally want more and more per-backend stats. I think what we'd want is rather something where for each applicable stats kind we have a shared counter for all exited backends and then per-backend counters for live backends, with helpers to aggregate the exited + live stats to a total.

Greetings,

Andres Freund

Live Demo

Mixed workload → identify the misbehaving backend



Thank you!

Questions?

`bdrouvot@amazon.com`
`https://bdrouvot.github.io`

