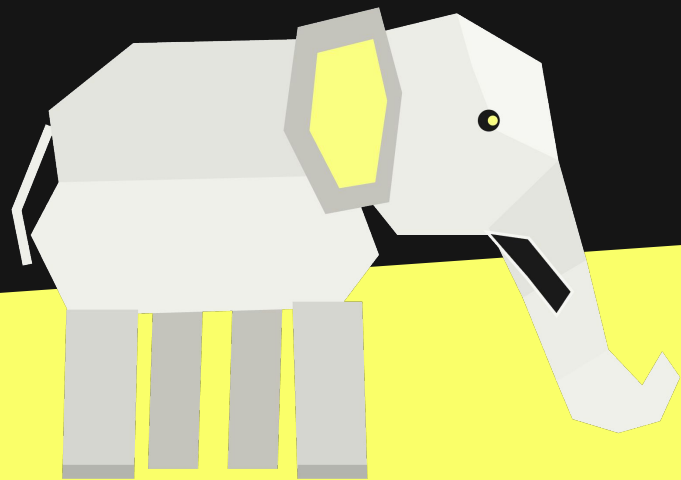


What's new with monitoring in PostgreSQL 19

PostgreSQL Observability Summit · PGConf.EU 2026

23 October 2026, Valencia

Gülçin Yıldırım Jelinek



whoami

Select * from me;

- Working with PostgreSQL since 2012
- PostgreSQL Europe board, 2017; PostgreSQL contributor, 2024
- Main organizer of Prague PostgreSQL Meetup
- Co-organizing **PostgreSQL Observability Summit**
- Co-organizing **PostgreSQL on Kubernetes Summit**

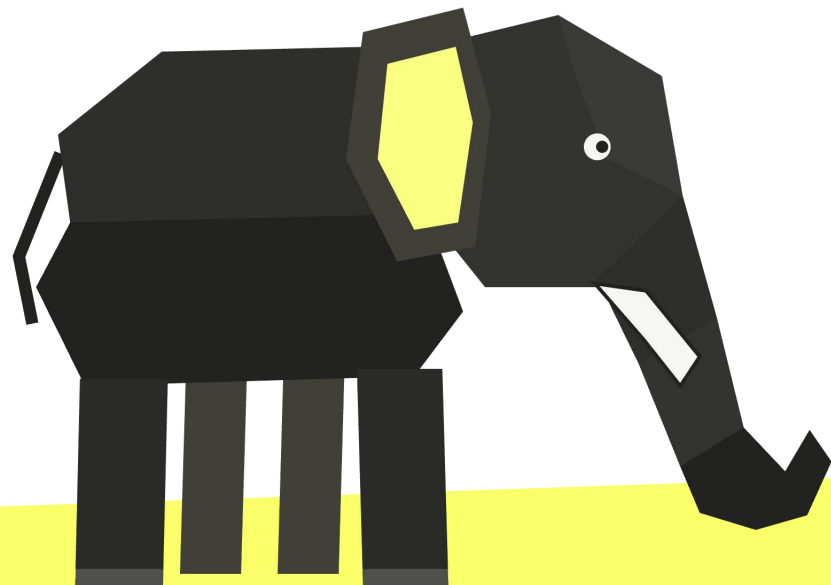


Gülçin Yıldırım Jelinek

ClickHouse Managed Postgres

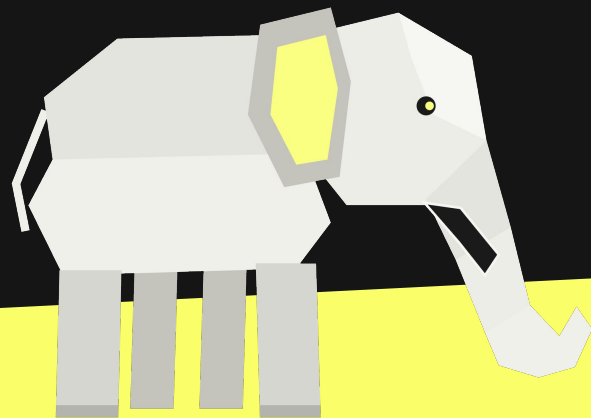
Agenda

- 01** Logging
- 02** WAL and I/O
- 03** WAIT FOR
- 04** New system views
- 05** Multixacts and wraparound
- 06** Future, 19 and beyond



01 Logging

Four changes to what Postgres writes to its log, and when



log_lock_waits is now on by default

Lock contention becomes visible without touching the config

- Logs a message when a session waits longer than `deadlock_timeout` (default 1s) to acquire a lock
- The check piggybacks on the deadlock detector, which is why the threshold is `deadlock_timeout` and not its own setting
- A cheap lock-contention detector that has shipped switched **off** since it was introduced — PostgreSQL 19 flips it **on**

"If someone is stuck behind a lock for more than a second, that is almost always a problem that is worth a log entry."

Laurenz Albe

After upgrading

Lock waits are logged only once they exceed `deadlock_timeout` (default 1 s), so that setting now also controls what reaches your log — check your value. Then alert on “still waiting for lock”: these lines are new on most clusters.

One timer, two consequences

deadlock_timeout decides what reaches your log and what pg_stat_lock counts



server log
(nothing yet)

pg_stat_lock		
locktype	waits	wait_time
relation	0	0.000

deadlock_timeout = 1 s, the wait lasts 2.2 s: at 1 s the deadlock detector wakes, finds no cycle, and log_lock_waits writes a line; pg_stat_lock counts the wait when the lock is granted

Different log verbosity per process type

log_min_messages accepts a list of type:level pairs (Euler Taveira)

```
-- DEBUG2 for the checkpointer, DEBUG1 for autovacuum, WARNING for everything else
ALTER SYSTEM SET log_min_messages = 'warning, checkpointer:debug2,
autovacuum:debug1';
```

- One bare level is mandatory, it applies to every process type not listed
- Fifteen process types: archiver, autovacuum, backend, bgworker, bgwriter, checkpointer, ioworker, postmaster, slotsyncworker, startup, sysloger, walreceiver, walsender, walsummarizer, walwriter
- Note autovacuum covers both the launcher and the workers

After upgrading

Nothing breaks for users: a single level is still valid. If you ship a config validator or UI that checks **log_min_messages**, teach it the **type:level** list syntax, or it will reject valid settings.

Autoanalyze logging split from autovacuum

New `log_autoanalyze_min_duration` (Shinya Kato)

Before

- `log_autovacuum_min_duration` covered both VACUUM and ANALYZE runs by autovacuum
- Analyze runs are much shorter, so a threshold tuned for slow vacuums silently discarded almost all analyze activity

PostgreSQL 19

- `log_autoanalyze_min_duration` controls ANALYZE; `log_autovacuum_min_duration` is VACUUM-only
- Both default to **10min**; **0** logs everything, **-1** disables
- Per-table: `ALTER TABLE events SET (log_autoanalyze_min_duration = 0);`

Before 19

If you parse autovacuum logs, set `log_autoanalyze_min_duration` explicitly. Otherwise ANALYZE runs silently stop appearing after the upgrade, because the old setting no longer covers them.

Remote messages, formatted like local ones

Replication, postgres_fdw and dblink messages go through ereport() (Vignesh C)

Before

- NOTICE / WARNING from a remote server were printed straight to the local server's stderr
- No **log_line_prefix**, no timestamp, no PID — hard to correlate with nearby log entries

PostgreSQL 19

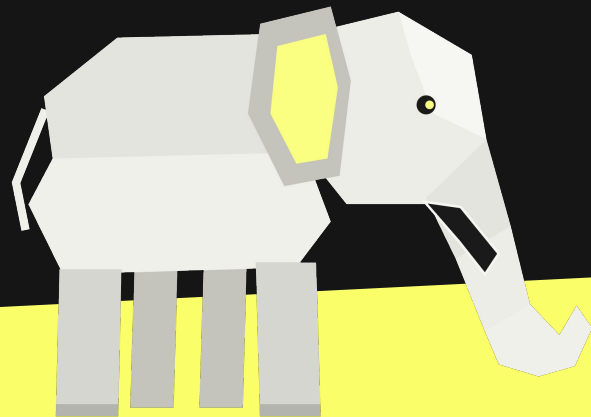
- Routed through ereport() like local messages: normal formatting and prefix
- Origin tag: **received message via replication** or **received message via remote connection**

After upgrading

Cross-server log correlation gets much easier. Remote-connection messages arrive at local LOG severity regardless of remote severity, so filter on message text, not severity.

02 WAL and I/O

The true I/O cost of maintenance, and a blind spot in COPY



Full-page images in sixty seconds

Why one tiny update can cost 8 kB of WAL

1

First change after a checkpoint

Postgres writes the entire 8 kB page into WAL, not just the delta. Crash recovery never has to trust a possibly torn page on disk.

2

A 50-byte update → ~8 kB of WAL

If it is the first touch of that page since the last checkpoint. Later changes to the same page log only deltas — until the next checkpoint resets the clock.

3

VACUUM is an FPI factory

It dirties many cold pages (dead tuples, hint bits, freezing). Vacuuming a big cold table can generate far more WAL than the data it changed.

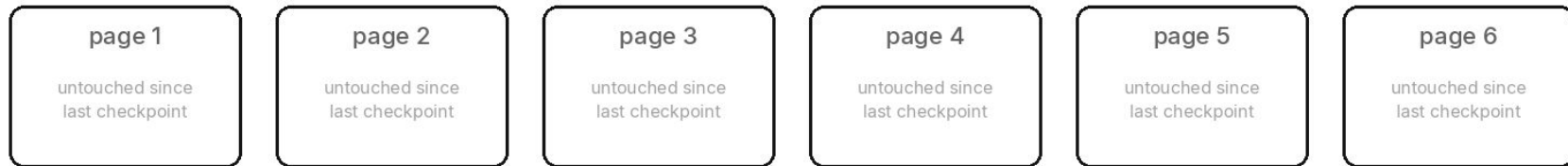
After upgrading

Postgres reported how many full-page images were written (`wal_fpi` in `pg_stat_wal`) but not their size. Multiplying by 8 kB is wrong: images may be compressed and unused page space is excluded. The real bytes needed `pg_waldump` or `pg_walinspect`.

One update, eight kilobytes

The first touch of a page after a checkpoint copies the whole 8 kB page into WAL

Shared buffers: six 8 kB pages



WAL



CHECKPOINT — every page is 'untouched' again

wal_fpi_bytes: the same number in four places

PostgreSQL 19 adds the bytes written as full-page images (Shinya Kato)

Cluster

`pg_stat_wal.wal_fpi_bytes`

Total since the last stats reset. This is what your metrics collector polls.

Session

`pg_stat_get_backend_wal(pid)`

The same counter for one backend. Which connection is producing the WAL?

Operation

`VACUUM / ANALYZE log line`

A new **“full page image bytes”** field in the WAL usage line of each run.

Query

`EXPLAIN (ANALYZE, WAL)`

A new **“fpi bytes”** value in the WAL line of the plan.

```
1 WAL usage: 2841 records, 1904 full page images, 15735621 bytes,
2           15242880 full page image bytes, 3 buffers full
```

What to do with it: the full-page-image ratio

```
SELECT wal_fpi_bytes::float / NULLIF(wal_bytes, 0) AS fpi_ratio FROM pg_stat_wal;  
-- what share of your WAL is full-page images?
```

High ratio

- Your WAL is dominated by first-touch-after-checkpoint page copies
- Levers: **wal_compression**, checkpoints further apart, scheduled maintenance

Low ratio

- Full-page images are not your problem
- Those knobs won't help; look elsewhere for the WAL volume

New I/O wait events for COPY

CopyFromRead and CopyToWrite (Nikolay Samokhvalov)

The gap

- COPY FROM / TO on a file, pipe or program did its reads and writes with no wait event at all
- A backend stuck in a bulk load looked idle in `pg_stat_activity` — `wait_event` was NULL
- COPY over the wire protocol was already covered by ClientRead / ClientWrite

PostgreSQL 19

- IO class: **CopyFromRead** — waiting to read from a pipe, file or program during COPY FROM
- IO class: **CopyToWrite** — waiting to write to a pipe, file or program during COPY TO
- Closes the paths commonly used by bulk loads and ETL pipelines

03 WAIT FOR

Three WAL stages on a standby, their wait events, and WAIT FOR

WAIT FOR: read your writes on async replicas

Wait until the standby has seen this write, then run the read

- Solves the stale read problem: write on the primary, read it straight back from a replica that has not caught up yet
- Old fixes: synchronous_commit = remote_apply (every commit pays), application-side polling on pg_last_wal_replay_lsn(), or read from the primary

```
-- on the primary
UPDATE movie SET genre = 'Dramatic'
  WHERE genre = 'Drama';
SELECT pg_current_wal_insert_lsn();
-- 0/0306EE20
```

```
-- on the standby
WAIT FOR LSN '0/0306EE20';
-- status: success
SELECT * FROM movie WHERE genre = 'Drama';
-- (0 rows)
```

After upgrading

Options go in a WITH clause: MODE 'standby_replay' (default), 'standby_write', 'standby_flush' or 'primary_flush', TIMEOUT '100ms', NO_THROW. The write stays asynchronous; only the read that needs the guarantee waits.

WAL on a standby: written → flushed → replayed

Each stage now has a wait event in pg_stat_activity (Xuneng Zhou)

Written

WaitForWalWrite

WAL received and written to the standby's WAL files; may still sit in OS buffers

Flushed

WaitForWalFlush

Durable on the standby's disk, but not yet visible to queries

Replayed

WaitForWalReplay

Applied by the startup process; reads on the standby can see the changes

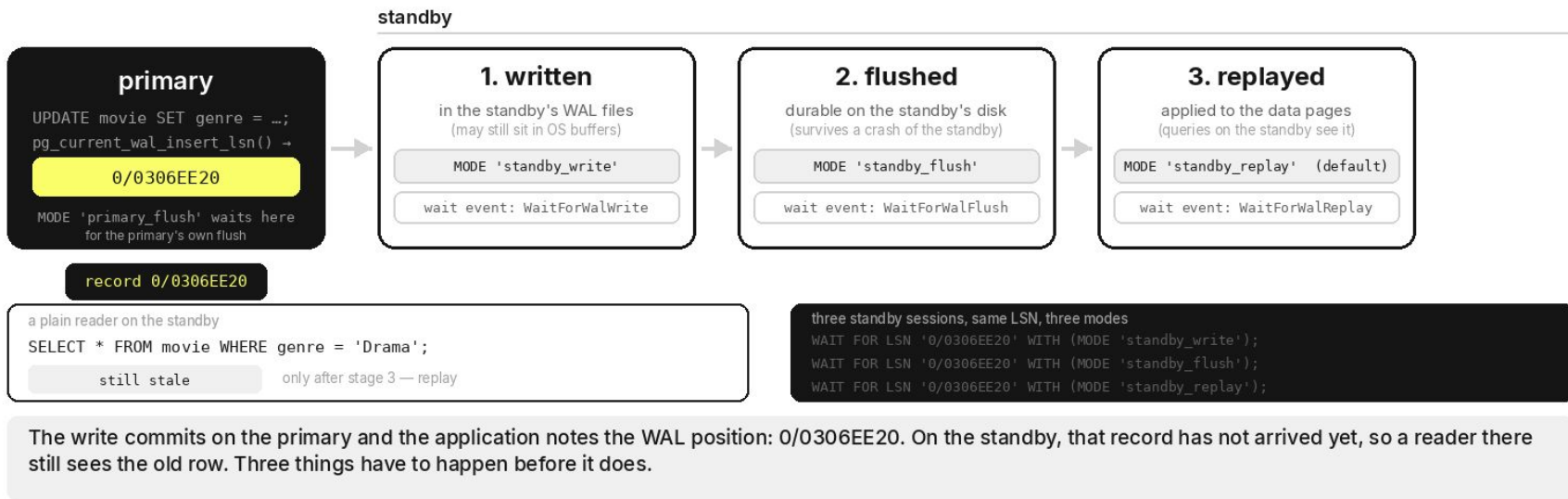
After upgrading

You can now tell which stage of WAL processing a session is waiting on — and the reason they exist at all is a new SQL command, next slide.

A write travels to the standby

Three stages on the standby, three wait events, three WAIT FOR modes

pass 1/5



04 New system views

`pg_stat_lock`, `pg_stat_recovery`, `pg_stat_autovacuum_scores` (and one more)

pg_stat_lock: lock waits, finally with history

Cumulative statistics per lock type (Bertrand Drouvot)

- Before: `pg_locks` (a snapshot of now, no history) or parse `log_lock_waits` output yourself
- One row per lock type, the kind of lockable object (relation, transactionid, tuple, extend, advisory, ...), not the lock mode
- Columns: `waits`, `wait_time` (ms), `fastpath_exceeded`, `stats_reset`; reset with `pg_stat_reset_shared('lock')`
- **Only counts locks acquired after waiting longer than `deadlock_timeout`**, it is not a counter of every lock wait

```
-- session 1: BEGIN; LOCK TABLE demo;
--                SELECT pg_sleep(2.5); COMMIT;
-- session 2: LOCK TABLE demo;  -- waits

SELECT locktype, waits, wait_time,
       round(wait_time::numeric / NULLIF(waits,0),
1)
      AS avg_wait_ms
FROM pg_stat_lock WHERE waits > 0
ORDER BY wait_time DESC;
```

locktype	waits	wait_time	avg_wait_ms
relation	1	2201.783	2201.8

fastpath_exceeded: locks that miss the fast path

The third counter in `pg_stat_lock`

What it counts

- Each backend keeps a small set of fast-path slots for the common locks that rarely conflict with anything
- When a query needs more locks than the slots hold, the extras fall back to the shared lock table (and this counter ticks, every time, no `deadlock_timeout` threshold)

Demo: a 140-partition table

- One `SELECT count(*)` over the parent locks the parent and every partition, during planning and execution
- `pg_stat_lock` afterwards: `relation | fastpath_exceeded = 422` the exact number varies between runs; what matters is that it is not zero

After upgrading

If it grows on a partition-heavy workload, raise `max_locks_per_transaction` (the slot count derives from it since 18). The new default of 128 only compensates for tighter lock-table sizing: an explicit setting now buys fewer locks than in 18, so double it.

pg_stat_recovery: standby state in one snapshot

Replaces four functions read at four different moments (Xuneng Zhou, Shinya Kato)

Before

- `pg_is_in_recovery()`,
`pg_last_wal_replay_lsn()`,
`pg_last_xact_replay_timestamp()`,
`pg_get_wal_replay_pause_state()`
- Each reads shared recovery state under its own lock, so the values were never guaranteed consistent with each other

PostgreSQL 19

- One row, read as a single atomic snapshot, all fields consistent
- Query it on a standby
- **pg_read_all_stats** privilege to see the data
- The old functions are still there

Autovacuum now has priorities

A new scoring system decides which table gets vacuumed first (Nathan Bossart)

Before PostgreSQL 19

- Autovacuum workers processed tables in the order they found them in `pg_class`
- A table one step from wraparound could wait behind a hundred barely-dirty ones

PostgreSQL 19

- Each worker scores every table: how close it is to (or how far past) its vacuum, insert, analyze, XID-age and multixact-age thresholds
- Highest score first — the tables that need attention most get it first

Tuning

Five `autovacuum_*_score_weight` settings, all defaulting to 1.0, weight the components. Setting all of them to 0.0 restores the old `pg_class` order. Nathan Bossart: ***"a baby step towards smarter autovacuum workers"***.

pg_stat_autovacuum_scores

What autovacuum will do next, table by table (Sami Imseih)

```
-- a 1000-row table, ANALYZEd, then 400 rows deleted
SELECT relname, round(score::numeric, 2) AS score, do_vacuum, do_analyze, for_wraparound
FROM pg_stat_autovacuum_scores
WHERE schemaname = 'public' AND (do_vacuum OR do_analyze)
ORDER BY score DESC;
```

relname	score	do_vacuum	do_analyze	for_wraparound
av_demo	2.67	t	t	f

- score = highest component: $400 \text{ changed rows} \div \text{analyze threshold } (50 + 0.1 \times 1000 = 150) = 2.67$; vacuum $400 \div 250 = 1.60$. A component above its weight (default 1.0) fires.

Caveat

A strong hint, not a guarantee: the view scores from current statistics, workers score when they wake up.

Who started this vacuum, and in which mode?

New columns in `pg_stat_progress_vacuum` and `pg_stat_progress_analyze` (Shinya Kato)

started_by

- **manual** — an explicit VACUUM or ANALYZE
- **autovacuum** — a worker; may be interrupted by a lock conflict
- **autovacuum_wraparound** — a worker preventing XID or multixact wraparound; not interrupted by lock conflicts (VACUUM view only)

mode (VACUUM view only)

- **normal** — a standard vacuum
- **aggressive** — must scan every page not marked all-frozen; triggered by `vacuum_freeze_table_age`
- **failsafe** — the minimum work to avoid wraparound; a vacuum can switch into this mid-run when `vacuum_failsafe_age` is reached

After upgrading

Alert on mode = 'failsafe' and on started_by = 'autovacuum_wraparound': those are the vacuums you must not cancel, and until 19 you could not tell them apart from routine ones in the progress view.

05 Multixacts

Multixact activity gets a function, and the wraparound warning arrives earlier

Multixacts in sixty seconds

What Postgres does when more than one transaction locks the same row

1

One xmax, many lockers

xmax has room for one transaction ID. When a second one locks the same row (FOR SHARE, FOR KEY SHARE, every foreign-key check), Postgres mints a multixact ID that stands for the list of lockers.

2

Where it lives

The ID sits in xmax with a flag; the member list (transaction ID + lock mode each) lives in pg_multixact/members, indexed by pg_multixact/offsets. Hot rows, many lockers: a big members area.

3

Its own wraparound

A 32-bit counter with its own age: relminmxid, mxid_age(), autovacuum_multixact_freeze_max_age (400M). VACUUM freezes old ones; members files go only once no table still needs them.

Why you care

The members area grows until the oldest multixact is frozen, so one long-running transaction, or a table autovacuum never reaches, keeps gigabytes on disk and pushes you toward multixact wraparound. Until 19 the only way to see it was the filesystem — next slide.

How a multixact is born, and why it lingers

Two lockers, one ID for the list, and why the members area shrinks late

one row version (orders, id = 42)

xmin	xmax	infomask	data
480	—	—	id=42 ...

transactions locking this row

pg_multixact/offsets
mxid → first member

pass 1/5

pg_multixact/members
member xid + lock mode, one entry each

members on disk: 0.0 GB

oldest multixact still needed (relminmxid): —

One transaction locks the row: a foreign-key check does `SELECT ... FOR KEY SHARE`. The locker's transaction ID goes straight into xmax, with a lock flag in the infomask. One slot, one transaction.

pg_get_multixact_stats()

Multixact activity was an observability blind spot (Naga Appani)

- Multixacts are created when several transactions lock the same tuple — foreign key checks, `SELECT ... FOR SHARE`
- Before: when `pg_multixact/members/` started eating disk, your options were filesystem inspection and educated guessing
- `num_mxids` spike → concurrent row-locking; `oldest_multixact` frozen while `num_members` grows → a long-running transaction blocks cleanup
- **Requires `pg_read_all_stats`**

```
SELECT *,  
    pg_size_pretty(members_size) AS pretty  
FROM pg_get_multixact_stats();
```

```
-[ RECORD 1 ]-----+-----  
num_mxids      | 311740299  
num_members    | 2785241176  
members_size   | 13926205880  
oldest_multixact | 2  
pretty         | 13 GB
```

More lead time before wraparound

Warning threshold raised for transaction IDs and multixact IDs (Nathan Bossart)

40M

Before

Remaining XIDs when Postgres started warning. At 10,000 XIDs per second that is about 67 minutes of runway.

100M

PostgreSQL 19

Warnings now start 2.5x earlier, to clients and to the server log. About 5% of the gauge instead of 2%.

3M

Unchanged

The stop limit, and wraparound itself, do not move. You just hear about it earlier.

After upgrading

If your `age(datfrozenxid)` alerts were calibrated against the old 40M warning point, recalibrate so they still fire before the server starts complaining.

Want more? Add to your watch list

Tue

WAIT FOR the Feature for Almost 10 Years

Alexander Korotkov

Wed

Stop Guessing: Unlocking Wait Events at Scale with Existing and New Tools

Dmitry Fomin

Wed

Per-Backend Statistics: See What Every Connection Is Actually Doing

Bertrand Drouvot

Thank you

Blog series:

- [What's new with monitoring in PostgreSQL 19](#)
- [Read your writes: WAIT FOR in PostgreSQL 19](#)
- [New system views in PostgreSQL 19](#)

