GSoC 2026 Proposal: Reducing pg_stat_statements LWLock Contention

PostgreSQL | Quan Hoang Truong | UMass Amherst

Contributor: Quan Hoang Truong

Email: truonghoangquan456@gmail.com

University: University of Massachusetts Amherst (Sophomore, CS)

Organization: PostgreSQL

Project Size: Large (~350 hours)

Mentors: Kirk Wolak, Nik Samokhvalov, Andrei Lepikhov

1. Project Description

pg_stat_statements is PostgreSQL's most widely used monitoring extension, tracking query statistics across a cluster. However, its internal locking architecture creates a scalability bottleneck: a single LWLock (`pgss->lock`) protects the entire shared hash table. Every query on the server must acquire at least a shared lock on this LWLock, and any structural modification (new entry insertion, deallocation of old entries, or reset) requires an exclusive lock that blocks all other backends.

Under workloads with high unique-query rates, the exclusive lock path in `pgss_store()` serializes backends. When `pg_stat_statements.max` is too small for the workload, constant deallocation via `entry_dealloc()` holds the exclusive lock while performing an O(n log n) sort, causing cascading stalls. In the worst documented case (Andrey Borodin, pgsql-hackers, 2022), this froze an entire production database.

This project proposes two core deliverables and two stretch goals. The core work is: (1) a pending-entry queue to avoid blocking on new entry insertion, and (2) a restructured deallocation critical section to reduce exclusive lock hold time. Both are independently shippable as small patches. The stretch goals are: (3) exploration of lock separation between structural changes and counter updates, and (4) an optimized reset path. This prioritization ensures a successful midterm and final evaluation even if the exploratory work proves too invasive for the GSoC timeline.

2. Problem Demonstration: Benchmark Results

I built PostgreSQL 19devel (HEAD) and 18.3 from source and ran benchmarks to quantify the contention. Setup: 16 vCPUs (WSL2), 64 clients, 20 pgbench scripts generating 1000 distinct normalized query forms, 60-second runs, compiled with `--enable-debug --enable-cassert --enable-injection-points`.

Configuration	Avg TPS	Deallocations	LWLock Waits	Overhead
No pgss (baseline)	106.81	n/a	0	-
pgss, max=5000	107.37	0	0	~0% (noise)
pgss, max=500	105.98	6,614	1,810	-0.8%
pgss, max=100	102.95	30,066	6,169	-3.6%

Key findings:

- At max=100 with high-frequency sampling (~50ms), **90-100% of active backends** are blocked on LWLock|pg_stat_statements at any given instant. At the kernel level, 44% of all samples are in

`futex_wait_queue` (the LWLock's underlying wait).

- The mechanism is `entry_dealloc()` at line 2176, which sorts all entries by usage and evicts the bottom 5% while holding the exclusive lock. With max=100 and 1000 unique forms, this fires ~30,000 times in 60 seconds.
- PG 19devel shows ~40% less overhead than PG 18.3 at max=100 (3.6% vs 6.3%), but the contention pattern is identical (~6,100 LWLock wait samples). PG19 holds the exclusive lock for less time per acquisition, but the fundamental single-lock bottleneck remains.
- I also tested `pg_stat_statements_reset()` under load (as suggested by mentor Kirk Wolak). Calling reset every 100ms produced 2,258 LWLock wait-event samples vs 0 without resets, confirming the same exclusive-lock stall pattern.

3. Proposed Solution

Core deliverables (committed for midterm and final evaluation):

Part 1: Pending-entry queue for new insertions

Currently, when `pgss_store()` encounters a new query ID, it releases the shared lock and blocks waiting for an exclusive lock, serializing all backends under contention. Proposed change: if the exclusive lock is busy, stash the new entry (query key + initial counters) in a small shared ring buffer. The next backend that acquires the exclusive lock drains the queue first. Entries are only lost if the queue fills up under sustained extreme contention. New `pg_stat_statements_info.queued` and `.dropped` counters make any data loss observable, addressing the main objection to Borodin's 2022 conditional-lock patch (which was rejected for silently dropping entries).

Part 2: Shorten the deallocation critical section

`entry_dealloc()` currently performs scan + qsort + eviction all under exclusive lock. The sort is O(n log n) and dominates the hold time.

Proposed change: snapshot usage values under shared lock, sort locally with no lock held, then take exclusive lock only for the actual `HASH_REMOVE` calls. Exclusive hold time drops from O(n log n) to O(n). Between snapshot and eviction, entries may have changed; we skip victims that no longer exist. The usage-based eviction is already approximate (comment at line 2191: "almost immediately obsolete"), so stale snapshots are acceptable.

Stretch goals (pursued after core deliverables are complete and benchmarked):

Part 3: Lock separation (exploration)

Currently both counter updates and structural changes go through `pgss->lock`. Separating these into two locks (one for structural modifications, one for counter updates) would allow the common path to proceed without competing with inserts or deallocs. The interaction between the two locks during dealloc (safely removing an entry that another backend might be updating) needs careful analysis. This is an exploration item: if benchmarks show the added complexity is not justified, the finding itself is a useful deliverable.

Part 4: Optimized reset

For full resets (no filter arguments): reduce time spent under the exclusive lock to near-constant by swapping in a fresh empty hash table under exclusive lock, then reclaiming old entries outside the critical section. For filtered resets: same snapshot-then-remove pattern as the dealloc optimization.

4. Alternatives Considered

- **Ring buffer (deferred writes):** Architecturally clean but requires a new shared memory structure, background worker, deferred visibility semantics, and merge logic. Too large a change for a first patch.

• **Sampling:** Changes the semantics of `calls` and `total_time`. Every monitoring tool that computes average latency (`total_time/calls`) would need to account for the sampling rate. Hard to sell to maintainers.

• **Partitioned hash table (dshash):** Worth investigating long-term and would help with new-entry insertion contention, but does not fully address dealloc or reset which still need to touch all partitions. The proposed fixes ship standalone and would still help even if dshash is adopted later.

5. Risks and Mitigations

The primary risk is that Part 3 (lock separation) may be too invasive or hard to reach consensus on. Mitigation: Parts 1 and 2 are prioritized as core deliverables and are independently valuable regardless of Part 3's outcome. If lock separation proves impractical, the analysis explaining why is itself a useful contribution.

6. Schedule (350 hours)

Period	Deliverables
Weeks 1-2	Formalize benchmark suite into a repeatable test harness. Add injection point tests for contention scenarios. Reproduce and document reset() contention. Baseline measurements on PG 18.3 and 19devel.
Weeks 3-4	Implement Part 1: pending-entry queue with LWLockConditionalAcquire(). Add queued/dropped counters to pg_stat_statements_info. Benchmark under load. Write regression tests.
Weeks 5-6	Implement Part 2: restructure entry_dealloc() to sort outside exclusive lock. Benchmark before/after. Analyze edge cases (concurrent dealloc, entry changes between snapshot and eviction). Write regression tests.
Midterm	**Reproducible benchmark harness delivered. Part 1 implemented and tested. Part 2 benchmarked patch with regression tests. Midterm evaluation.**
Weeks 7-8	Explore Part 3: lock separation prototype. Analyze the interaction between structural lock and counter-update lock during dealloc. Benchmark to determine if the added complexity is justified. Document findings either way.
Weeks 9-10	Implement Part 4: optimized reset() using structure-swap pattern. Race condition analysis and regression tests across all parts. Test with realistic workloads (mixed read/write, monitoring queries, periodic resets).
Weeks 11-12	Polish patches into one or more mailing-list-ready patch series. Final benchmark report comparing baseline vs each optimization. Write documentation of tradeoffs and rejected alternatives. Project summary and blog post.
Final	**Polished patches for Parts 1 and 2. Updated benchmarks. Mailing-list-ready writeup. Optional prototype for Part 3. Final evaluation.**

7. Expected Outcomes

Core (committed):

- A repeatable benchmark suite for measuring pgss LWLock contention under various workload profiles
- Patch implementing pending-entry queue for new entry insertion, with observability counters
- Patch restructuring entry_dealloc() to minimize exclusive lock hold time
- Regression tests for both patches
- Before/after performance benchmarks for each optimization
- One or more mailing-list-ready patch series formatted for pgsql-hackers review

- Benchmark report and documentation of tradeoffs and rejected alternatives

Stretch:

- Analysis of lock-separation feasibility with prototype and benchmark data (or documented rationale for why it is not viable)
- Optimized reset() path for full and filtered resets

8. Mentor Interaction

This proposal was shaped through direct discussion with the project mentors. I initially proposed a pure conditional-lock approach (skip recording when busy), but Kirk Wolak pointed out that pgsql-hackers rejected a similar 2022 patch for silently losing queries. That feedback led me to redesign Part 1 as a pending-entry queue. Kirk also suggested testing `pg_stat_statements_reset()` under load and exploring lock separation, both now in the proposal. Andrei Lepikhov advised starting with benchmarking, suggested injection points for reproduction, and confirmed that dshash is worth exploring long-term but nobody has compared it under high load, which informed my decision to focus on incremental fixes that ship standalone.

9. About Me

I am a sophomore CS student at UMass Amherst, currently interning at Deep Infra (AI cloud inference startup) where I patch and optimize open-source inference engines (vLLM, SGLang, TensorRT-LLM) for production, including profiling hot paths under high concurrency. I am also building a custom tokenizer based on HuggingFace's tokenizer library to reduce serving latency.

On the C side, I have a strong foundation from competitive programming in C++ but have not yet done large-scale C systems work in production. I have been reading the pg_stat_statements source and PostgreSQL's LWLock implementation comfortably, and growing into production-quality C is a major motivation for this project. I have been engaging upstream with vLLM and SGLang maintainers on patches; while none have merged yet, the experience taught me how open source communities make design decisions and that patience and alignment matter as much as code quality.

10. Availability and Post-GSoC

I am available full-time during the GSoC period (June-August 2026) with no other commitments. I am located in Massachusetts (EST) and comfortable with weekly Zoom calls and async communication via Telegram. After GSoC, I intend to continue contributing to PostgreSQL. I have joined the Postgres Hacking Telegram group and plan to participate in the vibe hacking sessions. My long-term goal is to build a career in low-level systems and database infrastructure.

11. References

- pg_stat_statements source: contrib/pg_stat_statements/pg_stat_statements.c
- Borodin incident report (2022): postgresql.org/message-id/1AEEB240-9B68-44D5-8A29-8F9FDB22C801@yandex-team.ru
- Rouhaud dshash suggestion: postgresql.org/message-id/20220912084047.c6fc26i2z5cwkaw7@jrouhaud
- Yhuel contention analysis (2021): yhuelf.github.io/2021/09/30/pg_stat_statements_bottleneck.html
- TantorLabs sampling analysis: dev.to/tantorlabs/redundant-statistics-slow-down-your-postgres-try-sampling-in-pgstatstatements-44em
- GSoC 2026 project idea: wiki.postgresql.org/wiki/GSoC_2026