PGCONF.DEV 2026 • CORE STORAGE & CONCURRENCY TRACK

Left-Right Lock (LRLock): Wait-Free Population-Oblivious Reads in Shared Memory

Author: **Greg Burd** | Source Tree Branch: `lrlck` RFC Prototype Target: PostgreSQL 19

Summary & Architectural Paradigm

The **Left-Right Lock (LRLock)** is a concurrency primitive developed for PostgreSQL that introduces entirely **wait-free, population-oblivious reads**. This scalability is achieved at the cost of maintaining two asymmetric copies of the underlying data structure in shared memory.

Unlike traditional reader-writer synchronization patterns, readers under LRLock proceed without acquiring any blocking lock or modifying shared state words. The read execution path is highly streamlined, consisting of a single atomic epoch counter increment, a full memory fence, and a pointer load.

Background: The shared LWLock Bottleneck

PostgreSQL's standard `LWLock` architecture relies on an atomic compare-and-swap (CAS) operation on a single lock state word. While highly efficient under low core counts, this model generates severe cache-line bouncing bottlenecks at high core counts (64+ cores) when hundreds of backends simultaneously vie for shared access (e.g., severe `ProcArrayLock` shared contention during statement-level `GetSnapshotData()` calls).

User-Space Quiescent Signaling

While the Linux kernel uses RCU (Read-Copy-Update) to separate read/write paths without reader synchronization, user-space shared memory lacked an equivalent mechanism. LRLock introduces per-reader epoch counters inside shared memory to serve as a high-speed quiescent state signaling system.

Core Algorithmic Properties Matrix

Design Matrix Attribute	LWLock (Shared Mode)	LRLock (Read Path)
Blocking Profile	Yes (CAS on shared state word)	**No (Wait-free protocol)**
Cache Line Contention	All backends contend on lock word	**Isolated to backend's epoch line**
Memory Cost Bounds	1x Data structure allocation	2x Data structure + padded epoch matrix
Write Cost Penalty	Single write under exclusive lock	Double-write + reader drain latency
Nested Reads	Supported (shared counter)	**Supported (nesting counter skip)**

Protocol State & Operation Mechanics

The synchronization loop ensures that writers never mutate a copy currently referenced by active readers, while readers always load a stable address.

READER & WRITER CONCURRENT EXECUTION PATHS

```
Reader (Wait-Free Path):                Writer (Serialized Path):
┌──────────────────────────┐            ┌──────────────────────────────┐
│ 1. Set bitmask bit       │            │ 1. Acquire writer mutex      │
│ 2. epoch++ (even->odd)   │            │ 2. Apply ops to write copy   │
│    [AcqRel]              │            │ 3. Memory barrier            │
│ 3. SeqCst fence          │            │ 4. Swap read_idx (0 ↔ 1)     │
│ 4. Load read_idx         │            │ 5. SeqCst fence              │
│    [Acquire]             │            │ 6. Snapshot epoch counters   │
│ 5. Read from             │            │    (bitmask-guided optimization)│
│    data[read_idx]        │            │ 7. Wait for pre-swap readers │
│ 6. epoch++ (odd->even)   │            │    to completely depart      │
│    [AcqRel]              │            │ 8. Replay oplog on stale copy│
│ 7. Clear bitmask bit     │            │ 9. Clear operation log (oplog)│
└──────────────────────────┘            │10. Release writer mutex      │
                                        └──────────────────────────────┘
```

Active-Reader Bitmask Optimization

Scanning a flat array of 1,000+ max_backends epoch lines per publish loop causes severe cache pressure (~64+ KB). LRLock avoids this via an atomic `pg_atomic_uint64` bitmask (typically 2-4 words).

The writer reads the bitmask words first, then exclusively fetches epoch values for backends with active flags set. Idle backends are skipped without touching their 64-byte padded epoch cache lines.

Per-Operation Log (Oplog) Engine

Instead of performing full-payload byte copies on large structures, modifications register via `LRLockApplyOp()`. Individual ops serialize into a growable buffer, allowing incremental updates (e.g., a 40-byte `ProcArrayXidOp` instead of a full 6 KB `ProcArray` snapshot copy).

If log accumulation grows dense (`oplog_count * 256 > data_size`), the writer automatically triggers a fallback sequential `memcpy` sync for speed.

Evaluated Engine Use Cases & Shared Memory Layouts

The LRLock primitive has been integrated and validated across three core transactional bottlenecks inside the PostgreSQL engine:

- **ProcArray Snapshot (GetSnapshotData):** Replaces the highly contested `ProcArrayLock` shared-mode path. Readers obtain a wait-free pointer to a `ProcArraySnapshotHdr` block. Standard modifications (commits/aborts) serialize via 40-byte operation entries, completely removing cache line bouncing under high backend concurrency.
- **Replication Slot Xmin Array:** `ReplicationSlotsComputeRequiredXmin()` is executed up to 14+ times per transaction under `ReplicationSlotControlLock` shared mode. LRLock replaces this with a compact 180-byte per-slot snapshot entry array (for typical configurations), enabling lock-free array sweeps.
- **Buffer Mapping Hash Table:** Replaces the upstream 128-way partitioned LWLock array structure with a single unified open-addressing hash table protected under LRLock epochs. This enables holding read guards safely across lookup and pin operations to eliminate Time-of-Check to Time-of-Use (TOCTOU) race conditions.

Empirical Performance Validation (SiFive 8-Core RISC-V)

Evaluated against three distinct pgbench profiles on a cache-coherent multiprocessor where atomic synchronization contention costs are high relative to standard x86 architectures (Scale factor 50, `fsync=off`, `max_connections=100`).

Workload Target Profile	Concurrent Clients	Master Engine (TPS)	LRLock Engine (TPS)	Throughput Delta
SELECT-only (Simple Protocol)	2 Clients	14,820	15,180	**+2.4% Gains**
SELECT-only (Simple Protocol)	4 Clients	27,530	28,250	**+2.6% Gains**
SELECT-only (Simple Protocol)	8 Clients	48,960	49,100	**+0.3% Gains**
TPC-B (Mixed Read/Write)	2 Clients	3,420	3,510	**+2.6% Gains**
TPC-B (Mixed Read/Write)	8 Clients	6,180	6,270	**+1.5% Gains**
TPC-B (Mixed Read/Write)	64 Clients	5,890	6,070	**+3.0% Gains**
SELECT-only (Prepared Protocol)	4 Clients	31,200	32,050	**+2.7% Gains**
SELECT-only (Prepared Protocol)	8 Clients	55,400	57,170	**+3.2% Gains**
SELECT-only (Prepared Protocol)	16 Clients	79,800	80,820	**+1.3% Gains**

** Analysis Summary: Maximum performance acceleration manifests under high read-heavy volumes at moderate concurrency bands, where lock shared-mode contention is eliminated before downstream scheduling or buffer resource caps saturate the infrastructure.*

Ecosystem Framework Status & Test Completeness

All 245 Core Regression subtests pass cleanly • Dedicated `test_lrlock` concurrent suite verified.
Valgrind execution testing profiles report absolute zero structural allocation leaks (67 processes, 0 errors).
Contact & Collaboration: **Greg Burd** <greg@burd.me>

Git Development Tree: `https://github.com/gburd/postgres/tree/lrlck`
Implementation Path: `src/backend/storage/lmgr/lrlock.c`