PGCONF.DEV 2026 • STORAGE ARCHITECTURE EXTENSION TRACK

Multiple Buffer Pools: Pluggable Replacement Cache Algorithms for PostgreSQL

Author: **Greg Burd** | Ecosystem Tree Branch: `arc` **contrib Suite Inside**

SUMMARY & ARCHITECTURAL PARADIGM

The **Multiple Buffer Pools** patch architecture extends PostgreSQL's storage buffer manager to partition `shared_buffers` into isolated, named memory regions. Each distinct pool runs completely under its own pluggable, workload-specific replacement algorithm vtable.

Database operators assign relations (tables, indexes, or specific fork descriptors) to targets using the new `buffer_pool` reloption. This breaks the historic system-wide reliance on a single unified 8-way clock-sweep routine, enabling specialized caching patterns for hot tables vs. volatile indexes.

The Pluggability Framework

The infrastructure wraps the default buffer manager inside a lightweight virtual function table (`vtable`) interface. The framework's core runtime path is localized down to a single indirect routine loop call (`ActivePoolRoutine->get_victim`) wrapped within upstream `StrategyGetBuffer()` execution bounds.

THE UNIFIED SQL DDL INTERFACE

Buffer pool lifecycles match core transactional system catalogs. Metadata persists natively inside the new `pg_bufferpool` system tracking frame:

```
-- 1. Create a 64 MB Adaptive Replacement Cache (ARC) Pool
CREATE BUFFER POOL hot_oltp HANDLER arc_pool_handler SIZE '67108864';

-- 2. Create a catch-all catch-budget REMAINDER Pool
CREATE BUFFER POOL leftovers HANDLER lru_pool_handler REMAINDER;

-- 3. Route a target relation explicitly to a specific Pool
ALTER TABLE order_line SET (buffer_pool = 'hot_oltp');

-- 4. Dynamic cleanup teardown paths
DROP BUFFER POOL hot_oltp;
```

* Pool resizing (`ALTER...SET SIZE`) flushes underlying dirty pages and safely maps a clean cold Dynamic Shared Memory (DSM) allocation array segment.

BUILT-IN SPECIALIZED POOL TYPES

Beyond custom user-space extension pluggability, three native algorithms ship inside the core engine array to intercept distinct database access workloads:

1. KEEP Pool

An explicit zero-eviction allocation framework. It guarantees that a critical, small, hot operational set (e.g., core system lookup tables) is pinned permanently in memory without ever triggering clock-sweep loops.

2. RECYCLE Pool

A single-chance, fast-discard clock routine. It completely intercepts large bulk operations (sequential scans, heavy analytical queries, `VACUUM` runs, or massive copy writes) without causing pollution across hot production caches.

3. JAM Pool

An accelerated-decay clock model optimized for the write-heavy, read-few transaction pattern typical of `TOAST` overflow storage bytes, ensuring rapid cache recycle rates.

Workload Overflow & TOAST Routing

The infrastructure introduces `overflow_buffer_pool` reloptions. If page allocation parameters cross their maximum bounds, large variable data attributes can gracefully overflow into the JAM pool while keeping the parent relation hot.

Each active user pool instantiates a dedicated **Trickle Writer** background worker. This unloads the main checkpointer loop, guaranteeing continuous dirty-page flushing tailored to the pool's specific algorithm eviction speed.

MICROBENCHMARK GRID: PLUGGABILITY FRAMEWORK OVERHEAD VERIFICATION

Evaluated using an exhaustive 19-pattern execution suite testing high-frequency lookups against a 100k-row tracking table enclosed inside a 128 MB `shared_buffers` layout. Cache hits are held locked at exactly 1.0 to isolate the true vtable indirection overhead from underlying disk behavior:

Configured Pool Extension	Mean Query Execution Step Speed (Seconds)	Relative Execution Slowdown vs. Classic Clock Baseline	Algorithmic Internal Cache Policy Character
`clock` (Core Native Upstream)	0.06703 s	*Baseline Standard*	Legacy 8-way sequential clock-sweep page replacement
`arc` (Adaptive Replacement Cache)	0.07436 s	**+10.9% Delta**	Dynamically balances recency and frequency tracking arrays
`lirs` (Low Inter-Reference Recency Set)	0.07525 s	**+12.3% Delta**	Maximizes execution efficiency on looping sequential sweeps
`car` (Clock with Adaptive Replacement)	0.07542 s	**+12.5% Delta**	Combines ARC advantages with low overhead clock structures
`lru` (Least Recently Used Baseline)	0.07565 s	**+12.9% Delta**	Classic standard recency queue tracking behavior
`osic` (One-Shot Internal Cache)	0.07582 s	**+13.1% Delta**	Custom specialized workload filter routine suite

** Framework Performance Analysis: The raw pluggability infrastructure cost is near zero (one indirect call per victim search). The slight delta metrics shown above are purely the result of each specific algorithm's internal tracking bookkeeping overhead.*

PRODUCTION MONITORING & CATALOG OBSERVABILITY ELEMENTS

Operators track real-time cache wellness vectors across dynamic shared segments using new dedicated management objects:

- **System Catalog: `pg_bufferpool`** — Stores core pool definition states, mapping unique pool identifiers directly to size budgets, routine extension entry handlers, and active attachment weights.
- **Statistics View: `pg_stat_bufferpool`** — Exposes real-time analytics including `alloc_buffers`, `dirty_pages_written`, `vtable_victim_calls`, and explicit hit/miss ratios separated per pool region. This lets DBAs precisely spot cache pollution or misallocated size budgets instantly.

Ecosystem Framework Status & Test Completeness

All 245 Core Regression components pass cleanly • Built-in `contrib` test matrix validated.
Integrated stats accounting verifies zero background worker leakages across pool boundary transitions.

Contact & Collaboration: **Greg Burd** <greg@burd.me>

Git Development Tree: `https://github.com/gburd/postgres/tree/arc`
Framework Catalogs: `pg_catalog.pg_bufferpool | pg_catalog.pg_stat_bufferpool`