BufBench: A Benchmarking Framework for Buffer Pool Analysis in PostgreSQL

Aroma Hoque and Tarikul Islam Papon

University of Massachusetts Boston, MA, USA
{aroma.hoque001, t.papon}@umb.edu

Abstract. The buffer pool manages all data movement between main memory and disk, making it one of the most central components of any database management system. Yet popular benchmarking tools like `pgbench` and `BenchBase` offer no visibility into buffer pool internals such as page eviction, buffer utilization, or background process interference. In this paper, we present BufBench, a purpose-built, open-source benchmarking framework that provides systematic analysis of PostgreSQL buffer pool behavior and its impact on overall performance. BufBench makes four core contributions. First, it provides a unified suite of six standard benchmarks: TPC-B, TPC-C, TPC-H, YCSB, CH-benCHmark, and TATP, covering a broad spectrum of OLTP, OLAP, skewed-access, and mixed workload characteristics. Second, it supports a pluggable page replacement policy interface, enabling direct comparison of PostgreSQL's default Clock-Sweep against alternative policies (currently including LRU, CFLRU, and LRU-WSR). Third, it incorporates a fine-grained buffer pool monitoring layer that captures 34 metrics per second from PostgreSQL's internal statistics views. Fourth, it automatically detects and quantifies the interference caused by checkpointing and vacuum events on throughput, a capability absent from all existing tools. Together, these capabilities make BufBench a ready-to-use diagnostic tool that database researchers and practitioners can deploy to analyze the whole design space of buffer pool behaviors.

Keywords: Buffer pool · Benchmarking · PostgreSQL · Page replacement · Performance evaluation · OLTP · OLAP

1 Introduction

In any relational database system, the buffer pool is a crucial component, as every data access flows through it [15,43]. The main goal of the buffer pool is to reduce expensive disk I/O by caching the pages with the highest chance of being accessed in the near future which improves the overall performance of the database management system [4]. Three factors jointly determine buffer pool effectiveness: the page replacement policy, the size of the buffer pool relative to the working set, and the interference caused by background processes such as checkpoints and autovacuum events. Without visibility into these factors and their interactions, database administrators are left blind, unable to reason about

Table 1: Comparison of existing Postgres benchmarking tools.

Feature	pgbench	BenchBase	pgbench-tools	BufBench
TPC-B	✓	✓	✓	✓
TPC-C		✓		✓
TPC-H		✓		✓
YCSB		✓		✓
CH-benCHmark		✓		✓
TATP		✓		✓
Buffer hit rate				✓
Eviction rate				✓
Buffer utilization				✓
Dirty buffer count				✓
Checkpoint interference				✓
Vacuum interference				✓
Page replacement policies				✓
PostgreSQL 18	✓			✓
Open source	✓	✓	✓	✓

sudden throughput drops, the true cost of background processes, or the behavior of alternative eviction policies.

Limitations of Existing Tools. Despite the buffer pool being the central engine of every relational DBMS, existing benchmarking tools provide no meaningful visibility into its internals. Table 1 shows that state-of-the-art tools such as `BenchBase` [9], `pgbench` [40], and `pgbench-tools` [45] share four critical gaps. First, none of them expose buffer pool metrics: hit rate, eviction rate, buffer utilization, and dirty buffer counts are universally absent, making it impossible to diagnose the root cause of performance changes. Second, background process interference from checkpoints and autovacuum, which can cause severe throughput drops [33], is not tracked by any tool. Third, no existing tool provides support for alternative page replacement policies, making controlled comparison of eviction strategies impossible despite decades of research on the topic [4,19,27,31]. Fourth, workload coverage remains fragmented: pgbench supports only TPC-B [49], while `BenchBase`, though broader [9], was never designed for buffer pool analysis and collects none of the metrics needed to evaluate it.

The Solution. To address these limitations, we present `BufBench`, an open-source benchmarking framework for systematic analysis of PostgreSQL buffer pool behavior, which makes four concrete contributions as we present below. We choose PostgreSQL since it is the most widely deployed open-source relational DBMS and its extensible statistics views make fine-grained instrumentation feasible. `BufBench` is publicly available at https://github.com/chill-umb/BufBench.

1. Six workloads implemented from scratch. We implement TPC-B [49], TPC-C [47], TPC-H [48], YCSB [7], CH-benCHmark [6], and TATP [30] in Python using `psycopg2`, without any dependency on external benchmarking frameworks. Each workload follows a consistent three-layer architecture: a schema

generator, a configurable data loader, and a multi-threaded transaction driver. The YCSB workload further supports user-defined read/write/scan/insert/delete mix ratios, allowing studying how access skew affects eviction behavior.

2. Pluggable page replacement policy interface. BufBench runs against a custom PostgreSQL 18.3 build that exposes an `eviction_algorithm` GUC parameter, enabling direct comparison of LRU [43], CFLRU [38], LRU-WSR [20], and the default Clock-Sweep under identical workload conditions. Prior work has proposed a range of eviction strategies for database buffer pools [4,19,27,31]; BufBench provides the infrastructure to evaluate them systematically.

3. Fine-grained buffer pool metrics. BufBench collects 34 buffer pool metrics per second from PostgreSQL's internal statistics views [39,41], covering hit rate, eviction rate, dirty buffer count, pinned buffer count, average page hotness, and checkpoint and autovacuum write activity, all written to a time-series CSV at configurable granularity.

4. Background process interference analysis. BufBench automatically detects checkpoint and autovacuum events and aligns them with the TPS timeline, computing per-event TPS drop percentage and recovery time and exporting results as both a structured CSV and an annotated timeline chart.

Key Findings. Using BufBench, we uncover buffer pool behaviors invisible to existing tools. We observe non-monotonic hit rate degradation in TPC-C and CH-benCHmark, where larger buffer pools paradoxically reduce hit rate due to checkpoint-induced displacement of the working set. We further show that the relative write behavior of replacement policies reverses between workloads: LRU and LRU-WSR reduce dirty writes below Clock-Sweep on TPC-B but increase them at small buffer sizes on TPC-C, demonstrating that policy conclusions drawn from a single workload cannot be safely generalized.

Paper Organization. Section 2 describes the architecture of BufBench. Section 3 describes each supported workload. Section 4 details the buffer pool metrics collected. Section 5 presents our experimental evaluation. Section 6 discusses related work. Section 7 concludes the discussion.

2 System Design

BufBench is a Python-based benchmarking framework where each experiment is driven by a single configuration file. A high-level overview of the architecture of BufBench is illustrated in Figure 1. It can be divided into five components that execute in a coordinated pipeline: (i) a *configuration layer* for setting the experimental parameters, (ii) a *schema and data loader* that creates and populates the target database, (iii) a multi-threaded *workload driver* to generate concurrent transactions, (iv) a *buffer pool monitor* to collect fine-grained metric values, and (v) a *reporter and interference analyzer* to quantify the impact of background processes on performance.

Configuration Layer. Each BufBench experiment is controlled by a complete and self-contained YAML configuration file. It specifies the PostgreSQL connection parameters, the workload type, scaling factor, client count, and run duration, as well as buffer pool settings such as buffer size and checkpoint and

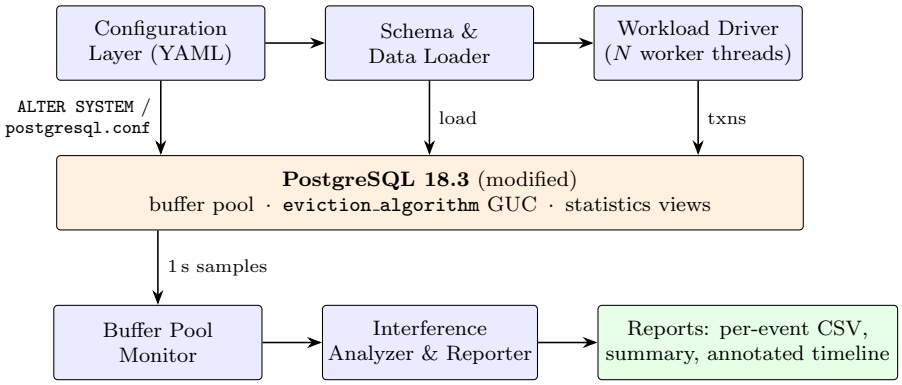


Fig. 1: Architecture of BufBench. Five components execute in a coordinated pipeline against a modified PostgreSQL 18.3 instance.

vacuum configuration. Before each run, BufBench applies these settings using `ALTER SYSTEM SET`, which writes them to PostgreSQL's `postgresql.auto.conf` file. For experiments targeting a modified PostgreSQL build with a custom page replacement policy, the active `eviction_algorithm` is configured directly in `postgresql.conf` and takes effect after a server restart.

Schema and Data Loader. The Python library `psycopg2` is used to connect to PostgreSQL and manage all database interactions. Data loading is performed using the `executemany()` interface of `psycopg2`, which batches multiple rows into a single network round-trip to significantly reduce loading time for large scale factors. By default, before each run the existing database is fully cleaned and the schema is recreated from scratch. If the database is already loaded for the correct workload and scale factor, for example, when varying only `shared_buffers` across a series of runs on the same dataset, BufBench supports a `--skip-setup` flag that bypasses schema creation and data loading entirely and moves directly to the benchmark run. BufBench ensures that the database matches the schema expected by the configured workload type when `--skip-setup` is active. If a mismatch is detected, the run is aborted immediately. The OS page cache is always flushed and the PostgreSQL buffer pool is always cleared via a server restart before the benchmark begins, ensuring a cold start for every run. After data loading is complete, all PostgreSQL statistics counters are reset using `pg_stat_reset()`, `pg_stat_reset_shared('bgwriter')`, and `pg_stat_reset_shared('checkpointer')`.

Workload Driver. Once the database is ready, N concurrent worker threads connect to the database via `psycopg2` and begin executing transactions sampled from a configurable workload distribution. For example, TPC-C [47] uses a 45% New-Order, 43% Payment, 4% Order-Status, 4% Delivery, and 4% Stock-Level mix. BufBench supports six benchmark workloads: TPC-B [49], TPC-C [47], TPC-H [48], YCSB [7], TATP [30], and CH-benCHmark [6], covering a broad spectrum of pure OLTP, pure OLAP, skewed-access, and mixed HTAP workload characteristics. Latency samples are accumulated in thread-local buffers and

flushed to a shared results queue in batches of 100 to minimize synchronization overhead. At the end of the run, all latency samples are merged and used to compute throughput (TPS), mean latency, and percentile latencies (p50, p95, p99). For CH-benCHmark and TPC-H, a separate pool of analytical threads runs concurrently alongside the OLTP worker threads for tracking per-query execution times. For TPC-H, Composite Query-per-Hour (QphH) [48] metric is calculated as the authoritative throughput metric.

Buffer Pool Monitor. Throughout the entire benchmark run, the buffer pool monitor runs as a dedicated background thread, sampling PostgreSQL's internal statistics views every second over a single autocommit connection. `pg_stat_database` provides cumulative buffer hit and miss counts to calculate the instantaneous hit rate. `pg_stat_bgwriter` provides the background writer's buffer cleaning activity, including the number of buffers cleaned and `maxwritten_clean` events, while `pg_stat_checkpointer` provides checkpoint activity such as timing, buffer write counts, and frequency. `pg_buffercache` [41] provides a real-time snapshot of every buffer slot in shared memory, from which buffer utilization, dirty buffer count, pinned buffer count, free buffer count, and average usage count are computed. In addition, `pg_stat_user_tables` is sampled to track autovacuum activity and dead tuple accumulation, which drives the vacuum interference detection described below. All 34 metrics are written to a timestamped CSV file on each sampling interval to produce a complete time-series record of buffer pool behavior. The full list of collected metrics is described in Section 4.

Background Process Interference Analyzer. Background processes such as checkpoints and autovacuum consume significant CPU and disk I/O, causing sudden TPS drops that can obscure genuine performance differences between buffer pool policies. As prior work has shown, basic throughput metrics alone are insufficient to reason about performance variability in storage systems [33]. To address this, BufBench includes an Interference Analyzer that automatically detects and quantifies the impact of these events on transaction throughput.

The analyzer reads the one-second-interval CSV produced by the Buffer Pool Monitor and scans for intervals in which the checkpoint or autovacuum counters increment, indicating a background event fired during that interval. For each detected event, it computes the pre-event TPS as the average of the three preceding seconds, the during-event TPS, and the resulting drop percentage. It also estimates the recovery time as the number of seconds until TPS returns to at least 90% of the pre-event baseline. Per-run-type aggregate statistics (total count, mean drop, maximum drop, and mean recovery time) are then computed and exported as three artifacts: a per-run CSV log, a human-readable text summary, and an annotated TPS timeline with checkpoint and vacuum events marked as vertical lines and labeled with their measured drop percentages.

Table 2 shows the interference report for a representative TPC-B run (15 GB database, 1500 MB buffer). The analyzer detected 10 checkpoint and 8 vacuum events over the 10-minute run. Nearly all events had negligible impact, with TPS recovering within one second. The average checkpoint drop remained well below 1%, and the single largest drop stayed under 2%. The one vacuum event near

Table 2: Interference report produced by BufBench for a TPC-B run (15 GB database, 1500 MB buffer, 600 s duration).

Time (s)	TPS Before	TPS During	Drop (%)	Recovery (s)
Checkpoint events				
57.7	609.0	617.0	0.0	1.1
117.5	617.3	609.0	1.3	1.0
176.2	629.7	621.0	1.4	1.0
234.0	647.3	653.0	0.0	1.0
291.7	652.7	646.0	1.0	1.0
348.4	660.7	649.0	1.8	1.0
405.1	663.3	666.0	0.0	1.0
461.8	683.7	672.0	1.7	1.1
517.6	699.0	692.0	1.0	1.0
573.3	695.7	706.0	0.0	1.0
Vacuum events				
40.6	609.0	617.0	0.0	1.0
100.3	611.3	618.0	0.0	1.0
161.0	622.7	630.0	0.0	1.1
220.8	631.7	650.0	0.0	1.0
280.6	644.3	653.0	0.0	1.0
340.3	661.0	662.0	0.0	1.0
401.1	654.0	681.0	0.0	1.0
520.6	692.0	677.0	2.2	1.0
Checkpoint avg (10 events)			**0.8**	**1.0**
Vacuum avg (8 events)			**0.3**	**1.0**

the end of the run produced a slightly larger dip coinciding with elevated write pressure, but recovered equally quickly. The annotated diagnostic plots for this run are presented and discussed in Section 5.3.

Page Replacement Policy Support. To evaluate alternative page replacement policies, BufBench targets a modified PostgreSQL 18.3 build. Stock PostgreSQL uses a Clock-Sweep algorithm [23] to evict pages from the buffer pool, which maintains a usage counter per buffer slot and performs a single pass to find an eviction victim [15,39]. The modified build exposes an `eviction_algorithm` GUC parameter, allowing the active policy to be switched between runs via `postgresql.conf` as described above. This design establishes a clean separation between policy evaluation and workload execution. Currently supported policies include Clock-Sweep, LRU [43], LRU-WSR [20], and CFLRU [38]. BufBench acts purely as the workload driver and metrics collector, while the replacement policy logic lives entirely within the PostgreSQL buffer manager, ensuring that performance differences between policies reflect only the eviction algorithm and not any instrumentation overhead.

Table 3: Summary of BufBench workloads.

Workload	Type	Tables	Txn Types	Access	Write %
TPC-B	OLTP	4	1	Random	100%
TPC-C	OLTP	9	5	Mixed	92%
YCSB	KV	1	6	Zipfian	Configurable
CH-benCHmark	HTAP	9	27	Mixed	92%
TPC-H	OLAP	8	5†	Sequential	0%
TATP	OLTP	4	7	Zipfian	20%

†Five representative queries selected from the 22 specified by TPC-H (see text).

3 Workload Descriptions

BufBench supports six workloads, each implemented as an independent Python class that inherits from a common base class. The base class handles connection management, worker thread coordination, and latency recording, while each subclass implements the workload-specific schema, data loader, and transaction driver. Adding a new workload to BufBench therefore requires implementing only these three components, without touching the monitoring or analysis layers. Together, the six workloads cover the full spectrum from pure write-heavy OLTP to pure analytical OLAP, with skewed-access and mixed HTAP workloads in between. Table 3 summarizes their key characteristics.

TPC-B. This write-intensive benchmark simulates a banking system with four tables: `branches`, `tellers`, `accounts`, and `history`. Every transaction touches all four tables, generating high dirty page counts and sustained checkpoint pressure. At scale factor 100 the `accounts` table alone reaches 1.28 GB, making `shared_buffers` a decisive performance parameter. TPC-B is the purest stress test for the write path of the buffer pool in BufBench.

TPC-C. This benchmark simulates a wholesale supplier using nine tables and five transaction types: New-Order (45%), Payment (43%), Order-Status (4%), Delivery (4%), and Stock-Level (4%). A single New-Order can touch up to eight tables in one commit. The mix of concurrent reads and writes across many tables produces diverse buffer pool access patterns that differ substantially from the uniform write pressure of TPC-B.

YCSB. This benchmark evaluates key-value workloads using a single `usertable` with ten string fields, supporting six operation types (read, update, insert, read-modify-write, scan, and delete) with user-defined mix ratios. Key selection follows a Zipfian distribution with a configurable skew parameter θ, concentrating accesses on a small hot set and directly shaping eviction behavior. YCSB is the only workload in BufBench where the read/write ratio is fully configurable.

CH-benCHmark. This HTAP benchmark runs the full TPC-C transaction mix on dedicated OLTP threads while all 22 TPC-H-style analytical queries execute concurrently on separate threads against the same database. Where necessary, queries are adapted to the TPC-C schema (e.g., Q17 is rewritten using a precomputed aggregate subquery to avoid correlated subquery issues at scale). The competition between analytical scans and OLTP transactions for

buffer pool slots makes CH-benCHmark uniquely valuable for studying buffer pool pressure under mixed workloads.

TPC-H. This decision support benchmark consists of eight tables and sequential full-table scans that flood the buffer pool with pages evicted after a single use, producing high eviction rates, low hit rates, and near-zero dirty buffer counts, a regime qualitatively different from all other workloads. We implement five representative queries (Q1, Q3, Q6, Q12, Q14) that cover the primary buffer pool stress patterns: full scans, multi-way joins, and selective filters. The remaining 17 queries stress the CPU, exercise the query optimizer, or involve correlated subqueries, making them less informative for buffer pool evaluation. CH-benCHmark retains all 22 queries to preserve HTAP fidelity.

TATP. This benchmark models a mobile network HLR database with four tables: `subscriber`, `access_info`, `special_facility`, and `call_forwarding`. Read operations account for 80% of the transaction mix, and a Zipfian distribution concentrates 80% of accesses on the top 20% of subscribers, creating a well-defined hot set. When `shared_buffers` holds this hot set, hit rates are high and performance is stable; otherwise, hot page evictions cause sharp throughput degradation, making TATP the cleanest workload for studying the relationship between buffer pool size and hit rate.

4 Buffer Pool Metrics

BufBench collects 34 fine-grained metrics at a fixed interval (one second by default, configurable in the YAML file) during the benchmark run using PostgreSQL's built-in statistics views [39,41]. These views are updated continuously by the PostgreSQL engine, providing a real-time picture of buffer pool internals. Table 4 lists all metrics organized by source view.

4.1 Key Metrics for Buffer Pool Analysis

The metrics reported in our experimental evaluation are divided into two categories: workload performance metrics that characterize throughput and buffer pool state under different configurations, and background process interference metrics that quantify the cost of checkpoint and autovacuum events.

Throughput, Latency and Buffer Hit Rate. Transactions per second (TPS) and average latency are the primary performance indicators, exposing trade-offs between buffer pool size and workload intensity [7,47,49]. The buffer hit rate, derived from PostgreSQL's `blks_hit` and `blks_read` counters [39], quantifies the direct impact of capacity constraints and eviction frequency on disk I/O. For mixed workloads like CH-benCHmark [6], aggregate averages can obscure transient drops caused by analytical queries displacing transactional pages. BufBench therefore collects all metrics as a per-second time series to render fine-grained buffer pool fluctuations visible.

Dirty Buffer Count. The dirty buffer count is the number of buffers that have been modified in memory but not yet flushed to disk, sampled every second from `pg_buffercache` [41]. It directly indicates the intensity of write pressure

Table 4: Buffer pool metrics collected by BufBench.

Source View	Metric	Description	Source View	Metric	Description
pg_stat_database	tps	Trans. per second	pg_stat_checkpointer	buffers_written	Pages at checkpoint
	commits	Committed txns		num_timed	Scheduled ckpts
	rollbacks	Rolled-back txns		num_requested	Forced ckpts
	blks_hit	Buffer pool hits		write_time	Write time (ms)
	blks_read	Pages read from disk		sync_time	Sync time (ms)
	hit_rate_pct	Hit rate (%)		checkpoint_happened	Event flag (0/1)
	tup_inserted	Rows inserted/s	pg_buffercache	total_buffers	Total slots
	tup_updated	Rows updated/s		used_buffers	Occupied slots
	tup_deleted	Rows deleted/s		free_buffers	Free slots
	tup_fetched	Rows fetched/s		dirty_buffers	Unflushed pages
pg_stat_bgwriter	buffers_clean	Pages evicted		buffer_utilization_pct	Utilization (%)
	buffers_alloc	Buffers allocated		avg_usage_count	Page hotness (0–5)
	maxwritten_clean	bgwriter pressure		pinned_buffers	Pinned buffers
	eviction_rate	Evictions/s	pg_stat_user_tables	autovacuum_count	Autovacuum runs
				dead_tuples	Dead rows (bloat)
				vacuum_happened	Event flag (0/1)
				heap_hit_rate_pct	Heap hit rate (%)
				index_hit_rate_pct	Index hit rate (%)

on the buffer pool. A persistently high count means that the buffer pool cannot absorb write bursts between checkpoints, which amplifies checkpoint interference and increases the cost of each checkpoint event. This effect is most pronounced in write-heavy workloads such as TPC-B [49] and TPC-C [47] at small buffer sizes, where the dirty buffer count approaches the total buffer pool capacity and checkpoint I/O competes directly with transaction processing.

Average Usage Count (Page Hotness). PostgreSQL's Clock-Sweep page replacement policy [15] maintains a `usage_count` for each buffer pool slot, incremented on every page reuse and decremented during eviction candidate scanning. BufBench computes the mean `usage_count` across all occupied buffer slots. This measure can be considered a proxy for overall page hotness. When this count remains significantly above 1.0, it indicates stability and reflects the repeatedly accessed working sets characteristic of OLTP workloads [4]. Conversely, when the count stays near or below 1.0, it reveals working set spillover, where the buffer pool is too small to retain hot pages, forcing the replacement policy to aggressively scan and evict pages.

Eviction Rate. `eviction_rate` measures the number of pages evicted by the background writer per second. It is derived as the per-second increment of the cumulative `buffers_clean` counter in `pg_stat_bgwriter` [39], computed by taking the difference between consecutive one-second snapshots. A sustained high eviction rate indicates that the buffer pool is under pressure and the background writer is actively evicting pages to make room for incoming data, which can compete with foreground transaction I/O, contributing to latency spikes.

Checkpoint Count and TPS Drop. BufBench detects a checkpoint event by monitoring the checkpoint counters in the `pg_stat_checkpointer` view [39], which increment during each checkpoint event. For each detected event, BufBench computes the TPS drop percentage as the relative decrease from the pre-event baseline TPS to the TPS observed during the event. We report both the average and maximum TPS drop across all checkpoint events: the average characterizes

typical overhead, while the maximum reveals worst-case spikes caused by a high dirty buffer count.

Vacuum Count and TPS Drop. To detect vacuum events, BufBench monitors the `autovacuum_count` counter in `pg_stat_user_tables`. Similar to checkpoint interference, BufBench computes the average and maximum TPS drop percentage caused by vacuum events. It additionally tracks `dead_tuples`, the total number of dead rows accumulated across all user tables, which provides a measure of table bloat and indicates how aggressively autovacuum needs to reclaim space. Write-heavy OLTP workloads such as TPC-B and TPC-C accumulate dead tuples rapidly and trigger frequent vacuum runs, whereas read-heavy or analytical workloads generate far fewer dead tuples and experience little interference, making vacuum impact workload-dependent. Reporting both average and maximum vacuum TPS drop allows practitioners to reason about performance variance and to distinguish between workloads.

4.2 PostgreSQL 18 Specific Metrics

BufBench is designed for PostgreSQL 18 and leverages two statistics improvements introduced in recent PostgreSQL versions. First, `pg_stat_checkpointer` (introduced in PostgreSQL 17) separates checkpoint statistics from the background writer, enabling precise measurement of checkpoint frequency, write time, and sync time independently of background writer activity [39]. Prior to this separation, checkpoint and background writer I/O were conflated in `pg_stat_bgwriter`, making it impossible to attribute write costs accurately. Second, `pg_buffercache` [41] in PostgreSQL 18 exposes the `pinning_backends` column, which counts the number of backend processes currently pinning each buffer, allowing BufBench to track actively accessed buffers in real time and distinguish pinned pages from merely cached ones.

5 Experimental Evaluation

We test our benchmarking tool BufBench across the six workloads at two database scales each. For every workload and scale, `shared_buffers` was varied across four sizes ranging from approximately 1% to 10% of the total database size, stressing the buffer pool under realistic memory pressure. The 48 experiments conducted are summarized in Table 5.

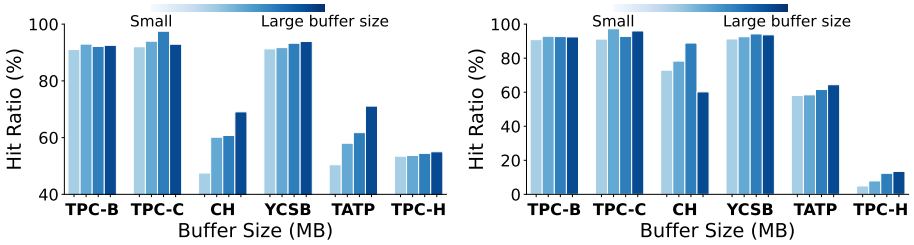
5.1 Experimental Setup

The experiments were conducted entirely on a ChameleonCloud [21] bare-metal node (Intel Skylake, 94 GB RAM, 210 GB NVMe SSD) running PostgreSQL 18.3 on Ubuntu 24.04. Each OLTP run executes for 10 minutes with 8 concurrent clients and each CH and TPC-H run executes for 20 minutes because the query execution times of the analytical workloads are longer. At the beginning of each test run, all tables are dropped to create a clean database state, which ensures a cold buffer pool at the beginning of every experiment.

Table 5: Throughput, latency, and hit rate across workloads and buffer sizes.

Workload	DB Size (GB)	Buffer Size (MB)	TPS	Avg Latency (ms)	Hit Rate Avg (%)	Usage Count (avg)
TPC-B	15	150	643.5	1.55	90.87	1.72
		450	635.9	1.57	92.86	1.45
		750	641.9	1.56	92.67	1.48
		1500	646.9	1.54	92.47	1.36
	7.5	100	684.4	1.46	91.03	1.78
		300	697.1	1.43	92.96	1.33
		500	686.9	1.45	92.14	1.35
		1000	685.1	1.46	92.52	1.33
TPC-C	4.3	50	148.7	6.72	92.01	1.72
		150	154.3	6.48	93.98	1.77
		250	142.6	7.01	97.49	2.05
		500	136.8	7.30	92.91	1.60
	8.3	100	114.6	8.73	91.21	1.38
		300	123.5	8.09	97.27	2.11
		500	116.8	8.56	92.77	1.61
		1000	120.9	8.26	95.98	1.80
CH-BenCH	4.4	50	164.4	6.08	47.51	0.73
		150	168.7	5.93	60.14	0.87
		250	165.7	6.03	60.75	0.91
		500	157.1	6.36	69.07	0.90
	8.4	100	134.2	7.45	72.94	0.81
		300	135.9	7.36	78.27	0.74
		500	124.1	8.06	88.89	0.77
		1000	129.6	7.72	60.19	0.81
YCSB	6.7	50	1872.6	0.53	91.28	1.29
		150	1920.3	0.52	91.72	1.31
		250	1939.4	0.51	93.26	1.44
		500	1960.1	0.51	93.87	1.53
	11	100	1850.7	0.54	91.29	1.27
		300	1875.4	0.53	92.58	1.33
		500	1894.3	0.53	94.24	1.48
		1000	1859.7	0.54	93.70	1.42
TATP	4.7	50	2258.1	0.44	50.42	0.81
		150	2235.4	0.45	58.01	0.71
		250	2257.5	0.44	61.74	0.65
		500	2264.2	0.44	71.10	0.80
	9.4	100	1981.8	0.50	58.07	0.80
		300	1978.6	0.50	58.45	0.71
		500	2000.8	0.50	61.57	0.65
		1000	2009.5	0.50	64.40	0.66
TPC-H†	4.5	50	2031	1774	53.43	1.19
		150	1905	1891	53.70	1.15
		250	1863	1934	54.44	1.12
		500	1848	1949	55.00	1.27
	10	100	711	5072	4.92	0.31
		300	747	4827	7.88	0.30
		500	756	4777	12.35	0.48
		1000	762	4729	13.47	0.46

†TPC-H reports Q/hour instead of TPS; Avg Lat is in ms per query.



(a) Cache Hit Rate (Small Scale) (b) Cache Hit Rate (Large Scale)

Fig. 2: Buffer pool cache hit ratio across workloads and buffer sizes.

5.2 Throughput, Latency, and Buffer Pool Hit Rate

Table 5 reports throughput, average latency, and buffer pool hit rate across all configurations. Figure 2 visualizes hit rate trends across both database scales. Workloads exhibit very different throughput characteristics reflecting transaction complexity. Read-heavy, index-driven workloads (TATP, YCSB) achieve the highest throughput via single-row primary-key lookups with minimal locking. TPC-B occupies a middle ground with a small fixed set of row updates per transaction. TPC-C and CH-benCHmark achieve considerably lower throughput due to multi-table joins and referential integrity enforcement. CH-benCHmark is further slowed by concurrent analytical queries competing for buffer pool resources. TPC-H reports query throughput and is not directly comparable.

Hit rate generally increases with buffer size. TATP and YCSB maintain high hit rates once the buffer holds the Zipfian hot set, while TPC-H stays persistently low due to sequential full-table scans that immediately displace cached pages. CH-benCHmark shows the widest variation, and both it and TPC-C exhibit a non-monotonic drop at the largest buffer size: as the buffer grows, PostgreSQL accumulates more dirty pages before a checkpoint fires, and flushing this enlarged dirty set displaces a significant fraction of the cached working set. This effect is amplified in write-heavy and mixed workloads, consistent with the high checkpoint counts and maximum TPS drops in Table 6.

5.3 Background Process Interference and Buffer Pool Behavior

Table 6 summarizes checkpoint and vacuum event counts and their worst-case configuration per workload. Figure 3 shows the annotated TPS timeline automatically generated by `BufBench` for a representative TPC-B run (15 GB database, 1500 MB buffer). All checkpoint drops were under 2% and recovered within one second, confirming that `checkpoint_completion_target = 0.9` spreads I/O smoothly. Vacuum events were similarly benign, with one exception near the end of the run coinciding with elevated write pressure.

Across workloads, checkpoint frequency is highest in CH-benCHmark and TPC-C, consistent with their higher dirty page generation rates. CH-benCHmark at the large scale shows the most severe maximum checkpoint drops, occasionally approaching near-total throughput suppression, which explains the hit rate anomaly in Figure 2. YCSB, TATP, and TPC-H trigger no vacuum events, as their transaction patterns produce little dead-tuple accumulation.

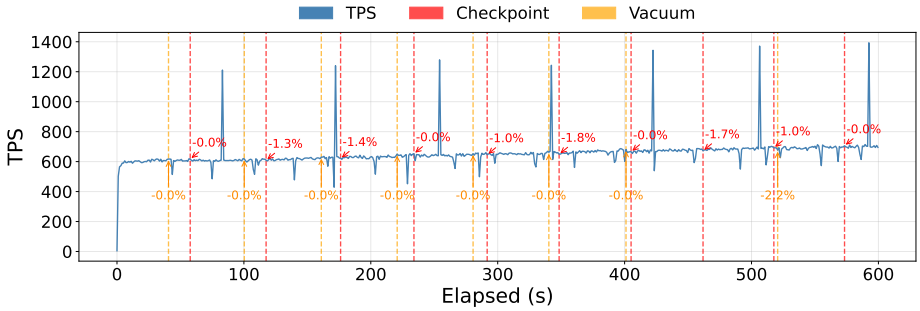


Fig. 3: Background process interference during a single TPC-B run (15 GB database, 1500 MB buffer). Red dashed lines mark checkpoint events; orange dashed lines mark vacuum events. Annotated percentages show the TPS drop relative to the pre-event baseline.

Table 6: Background process interference summary (worst-case configuration per workload). YCSB, TATP, and TPC-H trigger no vacuum events.

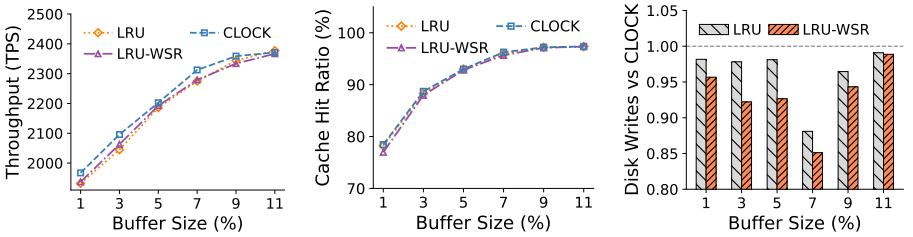
Workload	Ckpt Cnt	Ckpt Avg(%)	Ckpt Max(%)	Vac Cnt	Vac Avg(%)	Vac Max(%)
TPC-B	9	4.4	31.4	10	10.0	26.2
TPC-C	20	12.7	96.5	10	15.9	35.5
CH-BenCH	57	9.5	97.2	21	13.3	87.1
YCSB	10	12.8	27.2	0	–	–
TATP	5	7.4	32.2	0	–	–
TPC-H	4	26.2	39.8	0	–	–

5.4 Buffer Pool Replacement Policy Evaluation

Figures 4 and 5 compare CLOCK (PostgreSQL's default Clock-Sweep), LRU, and LRU-WSR on TPC-B (102 GB database) and TPC-C (50 GB database) respectively, with buffer size expressed as a percentage of the database size. BufBench also supports CFLRU; however, its two-pass candidate scanning causes severe lock contention under PostgreSQL's high-concurrency buffer manager, resulting in substantially higher latency ($3-4\times$) than all other policies, so we exclude it from this comparison.

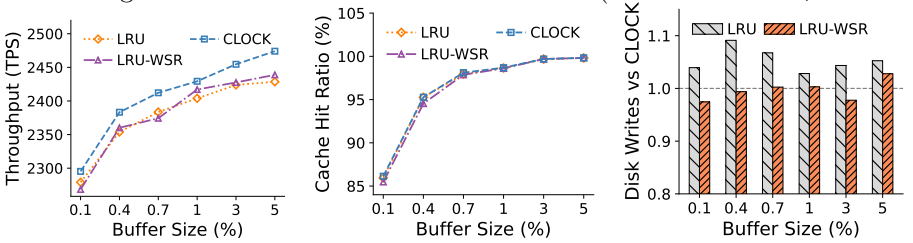
On TPC-B, all three algorithms deliver nearly identical throughput and cache hit ratios across all buffer sizes, confirming that the workload's sequential write pattern does not create meaningful differentiation between recency-based eviction strategies. The disk write panel tells a more informative story: both LRU and LRU-WSR consistently reduce dirty writes below CLOCK across all buffer sizes, with LRU-WSR achieving the larger reduction. This demonstrates that write-reduction benefits are measurable even when throughput differences are negligible, a distinction that aggregate-only benchmarks would miss entirely.

On TPC-C, throughput and hit rate again remain comparable across all three algorithms. As for disk writes, LRU yields higher writes than Clock across all buffer sizes because of the transactional nature of TPC-C, whereas LRU-WSR achieves a generally lower number of writes because of its write-awareness. As the buffer grows, the ratios converge and eventually fall below CLOCK. TPC-



(a) Throughput (b) Cache Hit Ratio (c) Disk Writes vs CLOCK

Fig. 4: Performance on TPC-B Benchmark (102 GB database).



(a) Throughput (b) Cache Hit Ratio (c) Disk Writes vs CLOCK

Fig. 5: Performance on TPC-C Benchmark (50 GB database).

C's higher write intensity and complex multi-table access pattern mean that at small buffers, recency-biased eviction displaces pages that are written again shortly after, increasing write amplification rather than reducing it.

5.5 Discussion

The results across six workloads and two database scales demonstrate that BufBench successfully captures the full diversity of buffer pool behavior under realistic PostgreSQL conditions. Buffer pool performance is not monotonically determined by buffer size alone: workload write patterns, checkpoint scheduling, and background process interference all interact to shape the observed hit rate and throughput. The non-monotonic hit rate drops in TPC-C and CH-benCHmark are a direct consequence of checkpoint behavior that only becomes visible through per-run time-series instrumentation. The reversal of dirty write ordering between TPC-B and TPC-C shows that replacement policy evaluation is highly workload-dependent, and that conclusions drawn from a single workload cannot be safely generalized. Together, these findings validate the need for a purpose-built benchmarking framework that exposes buffer pool internals alongside standard performance metrics.

6 Related Work

Database benchmarking has a rich history rooted in standardized workloads established by the Transaction Processing Performance Council (TPC), which have served as the foundation for database evaluation for several decades [47,48,49]. We organize related work into four categories: benchmarking frameworks, buffer

pool and storage analysis, HTAP and mixed workload benchmarking, and page replacement policy research.

6.1 Database Benchmarking Frameworks

pgbench [40] implements a simplified TPC-B [49] workload and reports throughput and latency percentiles, but is limited to a single workload with no buffer pool visibility. **pgbench-tools** [45] automates multiple pgbench runs across varying client counts but is equally restricted. **BenchBase** [9] supports over twenty workloads including TPC-C [47], TPC-H [48], CH-benCHmark [6], YCSB [7], and TATP [30], but collects no buffer pool metrics and performs no interference analysis. **HammerDB** [44] supports TPC-C and TPC-H style workloads across PostgreSQL, MySQL, and Oracle, but similarly exposes no buffer pool internals. **Tectonic** [33], presented at TPCTC 2025, bridges synthetic and real-world workloads for key-value storage engines such as RocksDB, with interference analysis focused on compaction rather than checkpointing. BufBench is complementary to Tectonic, targeting the relational buffer pool layer.

6.2 Buffer Pool and Storage Analysis

Buffer pool behavior has been studied extensively [3,12,24,29,34,37,42], yet no existing benchmarking tool exposes its internals in a systematic, workload-driven manner. Chou and DeWitt [4] demonstrated that replacement policy and buffer pool size jointly determine I/O efficiency. Sacco and Schkolnick [43] showed that even modest hit rate reductions produce disproportionate throughput degradation. These foundational results motivate the fine-grained hit rate, utilization, and eviction metrics that BufBench collects. Graefe [14] further noted that existing tools lack the instrumentation needed to evaluate replacement policies under varied workloads, a gap that BufBench addresses by exposing per-second eviction rates, dirty buffer counts, and average usage counts for rigorous comparison of Clock-Sweep, LRU, CFLRU [38], and LRU-WSR [20]. PostgreSQL's `pg_stat_checkpointer`, introduced in PostgreSQL 17 and leveraged fully in PostgreSQL 18, separates checkpoint I/O statistics from background writer activity [39], making it possible to attribute write amplification accurately which was previously impossible when both were conflated in `pg_stat_bgwriter`. BufBench is the first framework to use `pg_stat_checkpointer` for systematic, per-event interference measurement aligned with the transaction throughput time series.

6.3 HTAP and Mixed Workload Benchmarking

CH-benCHmark [6] combines TPC-C OLTP transactions with TPC-H analytical queries concurrently against the same database; Funke et al. [13] defined its performance metrics, including the queries-per-hour (QphH) metric for analytical throughput. HTAPBench [5] extended this with configurable OLTP-to-OLAP ratios, but like CH-benCHmark provides no buffer pool instrumentation. BufBench implements the full CH-benCHmark workload from scratch and goes further by tracking how concurrent analytical scans increase eviction pressure, degrade buffer hit rates, and displace hot OLTP pages, making it the first tool to expose the buffer pool cost of HTAP workloads at one-second granularity.

6.4 Page Replacement Policy

The question of which page to evict under buffer pool pressure has been studied for decades [12]. The classical LRU policy [4,43] remains the conceptual baseline. CLOCK [4,23] approximates LRU via a circular sweep with reference bits at lower overhead; PostgreSQL's Clock-Sweep is a practical implementation [15]. LRU-K [31,32] generalizes LRU by tracking the K most recent accesses per page to improve performance under sequential scans. The 2Q algorithm [19] uses separate queues for recent and frequent pages for scan resistance without manual tuning. ARC [27,28] dynamically balances recency and frequency queues, adapting to workload shifts without user-specified parameters. Several other policies have been proposed, including NFU [46], NRU [12], FIFO [46,50,51], LIRS [17], CLOCK-Pro [16], and Second Chance [22], each aiming to balance eviction quality and implementation overhead.

The widespread adoption of SSDs has motivated *flash-friendly policies* [36,37]. Since SSD writes are slower and increase device wear [1,8,10,35], these policies reduce writes by preferentially evicting clean pages and retaining dirty pages longer in memory [2,37]. CFLRU [38] targets write-asymmetric storage by preferentially evicting clean pages. LRU-WSR [20] extends this with a weak-strong reference distinction that protects frequently accessed pages from sequential scan flooding. Other flash-aware policies [11,18,25,26,37,52] adopt similar strategies considering access frequency and wear-leveling.

Despite this rich body of research, no existing benchmarking tool enables controlled, systematic comparison of these policies under standardized database workloads. BufBench fills this gap by exposing per-second hit rate, eviction rate, dirty buffer count, and average usage count across six standard workload types, providing the instrumentation necessary to evaluate both classical and novel page replacement policies in a reproducible experimental setting.

7 Conclusion

We introduced BufBench, an open-source benchmarking framework that enables systematic analysis of PostgreSQL buffer pool behavior. BufBench addresses four gaps left by existing tools: it unifies six standard workloads (TPC-B, TPC-C, TPC-H, YCSB, CH-benCHmark, and TATP), exposes a pluggable page replacement policy interface, collects 34 fine-grained buffer pool metrics per second, and automatically detects and quantifies the interference caused by checkpoints and autovacuum on transaction throughput. Our evaluation across 48 experiments reveals that buffer pool performance is shaped by the interplay of buffer size, workload write intensity, checkpoint scheduling, and eviction policy, all interactions that remain invisible to all existing tools. In particular, we observe non-monotonic hit rate behavior in write-heavy and mixed workloads, and show that replacement policy conclusions drawn from a single workload do not generalize across workloads. BufBench is publicly available and provides a ready-to-use diagnostic platform for researchers and practitioners studying the full design space of database buffer pool behavior.

References

1. Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J.D., Manasse, M., Panigrahy, R.: Design Tradeoffs for SSD Performance. In: Proceedings of the USENIX Annual Technical Conference (ATC). pp. 57–70 (2008), http://dl.acm.org/citation.cfm?id=1404019
2. Bagashvili, T., Papon, T.I., Athanassoulis, M.: ACE-in-Action: A Smart DBMS Bufferpool for SSDs. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. pp. 23–26 (2025), https://doi.org/10.1145/3722212.3725077
3. Chen, Q., Al-Araby, M.H.M., Qiu, Z., Chen, Z., Vinayak, R., Yang, J.: Demystifying and Improving Lazy Promotion in Cache Eviction. Proceedings of the VLDB Endowment **19**(4), 549–562 (2025), https://www.vldb.org/pvldb/vol19/p549-yang.pdf
4. Chou, H.T., DeWitt, D.J.: An evaluation of buffer management strategies for relational database systems. In: Proceedings of the International Conference on Very Large Data Bases (VLDB). pp. 127–141 (1985), http://dl.acm.org/citation.cfm?id=1286760.1286772
5. Coelho, F., Paulo, J., Vilaça, R., Pereira, J., Oliveira, R.: HTAPBench: Hybrid transactional and analytical processing benchmark. In: Proceedings of the International Conference on Performance Engineering (ICPE). pp. 293–304. ACM (2017)
6. Cole, R.L., Funke, F., Giakoumakis, L., Guy, W., Kemper, A., Krompass, S., Kuno, H.A., Nambiar, R.O., Neumann, T., Poess, M., Sattler, K.U., Seibold, M., Simon, E., Waas, F.: The mixed workload CH-benCHmark. In: Proceedings of the International Workshop on Testing Database Systems (DBTest). p. 8 (2011), http://doi.acm.org/10.1145/1988842.1988850
7. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the ACM Symposium on Cloud Computing (SoCC). pp. 143–154 (2010), http://doi.acm.org/10.1145/1807128.1807152
8. Cornwell, M.: Anatomy of a Solid-State Drive. Communications of the ACM (CACM) **55**(12), 59–63 (2012)
9. Difallah, D., Pavlo, A., Curino, C., Cudre-Mauroux, P.: OLTP-Bench: An extensible testbed for benchmarking relational databases. In: Proceedings of the VLDB Endowment. vol. 7, pp. 277–288 (2013)
10. Do, J., Zhang, D., Patel, J.M., DeWitt, D.J., Naughton, J.F., Halverson, A.: Turbocharging DBMS buffer pool using SSDs. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. pp. 1113–1124 (2011), http://dl.acm.org/citation.cfm?id=1989323.1989442
11. Dubs, P., Petrov, I., Gottstein, R., Buchmann, A.P.: FBARC: I/O Asymmetry Aware Buffer Replacement Strategy. In: Proceedings of the International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS). pp. 58–69 (2013)
12. Farooqui, M.Z., Shoaib, M., Khan, M.Z.: A comprehensive survey of page replacement algorithms. International Journal of Advanced Research in Computer Engineering & Technology (IJARCET) Volume **3** (2014)
13. Funke, F., Kemper, A., Krompass, S., Kuno, H.A., Nambiar, R.O., Neumann, T., Nica, A., Poess, M., Seibold, M.: Metrics for Measuring the Performance of the Mixed Workload CH-benCHmark. In: Proceedings of the TPC Technology Conference on Performance Evaluation, Measurement and Characteriza-

tion of Complex Systems (TPCTC). pp. 10–30 (2011), https://doi.org/10.1007/978-3-642-32627-1{_}2
14. Graefe, G.: Modern B-Tree Techniques. Foundations and Trends in Databases **3**(4), 203–402 (2011), http://dx.doi.org/10.1561/1900000028
15. Hellerstein, J.M., Stonebraker, M., Hamilton, J.R.: Architecture of a Database System. Foundations and Trends in Databases **1**(2), 141–259 (2007), http://dx.doi.org/10.1561/1900000002
16. Jiang, S., Chen, F., Zhang, X.: CLOCK-Pro: An Effective Improvement of the CLOCK Replacement. In: Proceedings of the USENIX Annual Technical Conference (ATC). pp. 323–336 (2005), http://www.usenix.org/events/usenix05/tech/general/jiang.html
17. Jiang, S., Zhang, X.: LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In: Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS). pp. 31–42 (2002), https://doi.org/10.1145/511334.511340
18. Jin, P., Ou, Y., Härder, T., Li, Z.: AD-LRU: An efficient buffer replacement algorithm for flash-based databases. Data Knowl. Eng. **72**, 83–102 (2012)
19. Johnson, T., Shasha, D.: 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In: Proceedings of the International Conference on Very Large Data Bases (VLDB). pp. 439–450 (1994), http://dl.acm.org/citation.cfm?id=645920.672996
20. Jung, H., Shim, H., Park, S., Kang, S., Cha, J.: LRU-WSR: integration of LRU and writes sequence reordering for flash memory. IEEE Trans. Consumer Electron. **54**(3), 1215–1223 (2008)
21. Keahey, K., Anderson, J., Zhen, Z., Riteau, P., Ruth, P., Stanzione, D., Cevik, M., Colleran, J., Gunawi, H.S., Hammock, C., Mambretti, J., Barnes, A., Halbach, F., Rocha, A., Stubbs, J.: Lessons Learned from the Chameleon Testbed. In: Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20). USENIX Association (jul 2020)
22. Kedzierski, K., Moretó, M., Cazorla, F.J., Valero, M.: Adapting cache partitioning algorithms to pseudo-LRU replacement policies. In: Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS). pp. 1–12 (2010), https://doi.org/10.1109/IPDPS.2010.5470352
23. Khuri, S., Hsu, H.C.: Visualizing the CPU scheduler and page replacement algorithms. In: Proceedings of the SIGCSE Technical Symposium on Computer Science Education (SIGCSE). pp. 227–231 (1999), https://doi.org/10.1145/299649.299764
24. Leis, V., Haubenschild, M., Kemper, A., Neumann, T.: LeanStore: In-Memory Data Management beyond Main Memory. In: Proceedings of the IEEE International Conference on Data Engineering (ICDE). pp. 185–196 (2018)
25. Li, Z., Jin, P., Su, X., Cui, K., Yue, L.: CCF-LRU: a new buffer replacement algorithm for flash memory. IEEE Trans. Consumer Electron. **55**(3), 1351–1359 (2009)
26. Lv, Y., Cui, B., He, B., Chen, X.: Operation-aware buffer management in flash-based systems. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. pp. 13–24 (2011), https://doi.org/10.1145/1989323.1989326
27. Megiddo, N., Modha, D.S.: ARC: A self-tuning, low overhead replacement cache. In: Proceedings of the USENIX Conference on File and Storage Technologies (FAST). pp. 115–130 (2003), http://www.usenix.org/event/fast03/tech/full_papers/megiddo/megiddo.pdf

28. Megiddo, N., Modha, D.S.: Outperforming LRU with an Adaptive Replacement Cache Algorithm. Computer **37**(4), 58–65 (2004), https://doi.org/10.1109/MC.2004.1297303
29. Neumann, T., Freitag, M.J.: Umbra: A Disk-Based System with In-Memory Performance. In: Proceedings of the Conference on Innovative Data Systems Research (CIDR) (2020), http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf
30. Neuvonen, S., Wolski, A., Manner, M., Raatikka, V.: TATP benchmark description. http://tatpbenchmark.sourceforge.net/ (2011)
31. O'Neil, E.J., O'Neil, P.E., Weikum, G.: The LRU-K Page Replacement Algorithm For Database Disk Buffering. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. pp. 297–306 (1993), https://doi.org/10.1145/170035.170081
32. O'Neil, E.J., O'Neil, P.E., Weikum, G.: An Optimality Proof of the LRU-K Page Replacement Algorithm. Journal of the ACM **46**(1), 92–112 (1999)
33. Ott, A., Kaushik, S., Chen, B., Sarkar, S.: Tectonic: Bridging synthetic and real-world workloads for key-value benchmarking. In: Proceedings of the TPC Technology Conference (TPCTC). LNCS, Springer (2025)
34. Papon, T.I.: Enhancing Data Systems Performance by Exploiting SSD Concurrency & Asymmetry. In: Proceedings of the IEEE International Conference on Data Engineering (ICDE). pp. 5644–5648 (2024), https://doi.org/10.1109/ICDE60146.2024.00454
35. Papon, T.I., Athanassoulis, M.: A Parametric I/O Model for Modern Storage Devices. In: Proceedings of the International Workshop on Data Management on New Hardware (DAMON). pp. 2:1–2:11 (2021), https://doi.org/10.1145/3465998.3466003
36. Papon, T.I., Athanassoulis, M.: The Need for a New I/O Model. In: Proceedings of the Conference on Innovative Data Systems Research (CIDR) (2021)
37. Papon, T.I., Athanassoulis, M.: ACEing the Bufferpool Management Paradigm for Modern Storage Devices. In: Proceedings of the IEEE International Conference on Data Engineering (ICDE). pp. 1326–1339 (2023)
38. Park, S.Y., Jung, D., Kang, J.U., Kim, J., Lee, J.: CFLRU: a replacement algorithm for flash memory. In: Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES). pp. 234–241 (2006)
39. PostgreSQL: PostgreSQL: The World's Most Advanced Open Source Relational Database. https://www.postgresql.org (2023)
40. PostgreSQL Global Development Group: pgbench — PostgreSQL benchmarking tool. https://www.postgresql.org/docs/current/pgbench.html (2024)
41. PostgreSQL Global Development Group: pg_buffercache — inspect PostgreSQL buffer cache contents. https://www.postgresql.org/docs/current/pgbuffercache.html (2024)
42. Ramakrishnan, R., Gehrke, J.: Database Management Systems. McGraw-Hill Higher Education, 3rd edition (2002)
43. Sacco, G.M., Schkolnick, M.: Buffer management in relational database systems. ACM Transactions on Database Systems (TODS) **11**(4), 473–498 (1986), http://dl.acm.org/citation.cfm?id=7239.7336
44. Shaw, S., Bond, A.: Hammerdb: From personal project to open benchmarking standard. In: Nambiar, R., Poess, M. (eds.) Performance Evaluation and Benchmarking - 17th TPC Technology Conference, TPCTC 2025, London, UK, September 1, 2025, Revised Selected Papers. pp. 134–154. Lecture Notes in Computer Science, Springer (2025), https://doi.org/10.1007/978-3-032-18070-4_9

45. Smith, G.: pgbench-tools. https://github.com/gregs1104/pgbench-tools (2024)
46. Tanenbaum, A.S.: Modern Operating Systems. Prentice-Hall (1992)
47. TPC: Specification of TPC-C benchmark. http://www.tpc.org/tpcc/
48. TPC: Specification of TPC-H benchmark. http://www.tpc.org/tpch/
49. TPC: TPC-B benchmark. https://www.tpc.org/tpcb/ (2021)
50. Yang, J., Qiu, Z., Zhang, Y., Yue, Y., Rashmi, K.V.: FIFO can be Better than LRU: the Power of Lazy Promotion and Quick Demotion. In: Proceedings of the 19th Workshop on Hot Topics in Operating Systems, HOTOS. pp. 70–79 (2023), https://doi.org/10.1145/3593856.3595887
51. Yang, J., Zhang, Y., Qiu, Z., Yue, Y., Vinayak, R.: FIFO queues are all you need for cache eviction. In: Proceedings of the ACM Symposium on Operating Systems Principles (SOSP). pp. 130–149 (2023), https://doi.org/10.1145/3600006.3613147
52. Yoo, Y.S., Lee, H., Ryu, Y., Bahn, H.: Page Replacement Algorithms for NAND Flash Memory Storages. In: Proceedings of the International Conference on Computational Science and Applications (ICCSA). pp. 201–212 (2007)