Making Array-Based Translation Practical for Modern, High-Performance Buffer Management

To appear at SIGMOD 2027

Xinjing Zhou
MIT CSAIL
Cambridge, MA, USA

Jinming Hu
Unaffiliated
China

Andrew Pavlo
Carnegie Mellon University
Pittsburgh, PA, USA

Michael Stonebraker
MIT CSAIL
Cambridge, MA, USA

Abstract

Modern buffer pools need to support a broader workload mix than just classic OLTP. Beyond B-tree lookups, database systems serve scan-heavy analytics and vector-search indexes with irregular, high-fan-out graph traversals. These workloads necessitate a page translation mechanism (i.e., mapping logical page IDs to resident frames) that is fast across diverse access patterns, deployable in user space without kernel modification, compatible with huge pages, easy to integrate into existing systems, and under DBMS control for eviction and fine-grained scalable I/O. Existing designs based on hash tables or OS page tables satisfy only subsets of these properties.

This paper presents **Calico** to address these challenges. Calico is a DBMS-controlled buffer pool built on array-based translation, a decades-old, dismissed idea that is now viable on modern hardware. Calico decouples logical translation from OS page tables, allowing the DBMS to combine low-overhead array translation with huge-page-backed frames and fine-grained page management. To make array translation practical and performant for modern workloads and DBMSs with large, sparse page identifiers, Calico introduces three key techniques: multi-level translation with path caching to accelerate lookups while managing sparsity, hole punching to reclaim cold translation memory, and group prefetch to exploit parallelism. These techniques allow Calico to achieve a balance of high-performance and manageability.

Our evaluation across OLTP-style B-tree accesses, vector search, and scans shows that Calico outperforms the state-of-the-art buffer pool designs while maintaining the desired properties. We also implement Calico as a drop-in replacement for PostgreSQL. Relative to PostgreSQL's hash table buffer manager, Calico delivers up to 3.95× in-memory and 6.57× larger-than-memory HNSW throughput, reduces IVFFlat median latency by up to 27.3%, and accelerates sequential scans and join queries, reaching 1.52–2.99× speedups.

Keywords

Database Management Systems, Caching, Buffer Management, Vector Search, OLAP, OLTP

ACM Reference Format:
Xinjing Zhou, Jinming Hu, Andrew Pavlo, and Michael Stonebraker. 2027. Making Array-Based Translation Practical for Modern, High-Performance Buffer Management: To appear at SIGMOD 2027. In *Proceedings of the ACM SIGMOD/PODS International Conference on Management of Data (SIGMOD-/PODS '27), June 13–19, 2027, Huntington Beach, CA, USA.* ACM, New York, NY, USA, 16 pages.

1 INTRODUCTION

Disk-oriented database management systems (DBMSs) support data sets that exceed available physical memory. A DBMS's buffer pool provides the key abstraction to achieve this functionality. It transparently handles the caching and eviction of pages in memory. When the DBMS's internal components (e.g., query executors) access data via logical page identifiers, they use the buffer pool to retrieve an in-memory buffer frame for the requested page. Thus, the core operation of a buffer pool is *translation*: mapping a logical page ID to an in-memory frame. Translation sits on the critical path of nearly every page access, so a buffer pool must keep it fast while retaining fine-grained DBMS control over page I/O and eviction.

Although buffer pool design and implementation is an old topic in databases [17], there are recent proposals for high-performance buffer pools [40, 42, 58, 90, 93]. These newer designs primarily target OLTP-style workloads, which are dominated by B-tree traversals. The rise of retrieval-augmented generation and semantic search is pushing DBMSs also to support vector search [59, 65, 75]. Such workloads have access patterns that impose significantly more stress on a buffer pool than OLTP workloads [47, 86]: (1) partition-based indexes are scan-heavy [10, 31, 38] and (2) graph-based indexes perform irregular, high-fan-out traversals [19, 30, 50]. Hence, a modern buffer pool must provide fast access for scan-heavy, B-tree, and graph workloads, and allow easy integration with data structures to support future workload patterns.

The most salient design choice in a buffer pool implementation is whether the DBMS or OS manages the page table(s) [12]:

DBMS-managed: The most common approach is for the DBMS to maintain the buffer pool's page table in user-space. Most production DBMSs implement the buffer-page translation table as a user-space hash table that maps logical page identifiers to buffer frames [8, 59, 66, 74]. This design is popular because it preserves deployability and keeps eviction and I/O policy under DBMS control, but it introduces significant costs such as table probing, pointer chasing, and synchronization, while hash functions also inherently scatter adjacent page IDs. As a result, scan-heavy workloads lose prefetch efficiency on modern CPUs, and high-parallelism graph workloads suffer from reduced memory-level parallelism. Predictive translation can improve hash-table translation [93], but it is workload-dependent and still cannot hide the translation overhead for the scan-heavy and graph workloads. Pointer swizzling can remove software translation [20, 42, 58], yet it is invasive to data-structure internals and difficult to apply to graph-style pages with variable numbers of incoming references.

OS-managed: An alternative is to store translation state in OS-managed hardware page tables [40, 90], where the TLB accelerates

resident-page lookup. However, this couples the DBMS with OS page-table management, weakens control over eviction and I/O scheduling, and requires kernel changes for scalable page fault handling [40, 90]. A further structural limitation is the *page granularity mismatch*: DBMSs need fine-grained page I/O (4–32KB), yet huge pages (2MB) are essential for TLB efficiency. Because the OS cannot cleanly evict a single 4KB subpage from a mapped 2MB huge page [2, 6, 33], these systems must choose between 4KB pages with higher TLB pressure or 2MB pages with larger I/O amplification.

Our experimental analysis across scans, B-tree lookups, and graph traversal shows that no single existing mechanism is preferable for these access patterns while also supporting huge pages and fine-grained DBMS I/O control. These limitations motivate a design goal that combines the strengths of both families: retain the full policy control of DBMS-managed translation state (eviction, recovery, and I/O scheduling) while approaching the translation efficiency of OS-managed page-table-based mechanisms without invasive changes to data structures or the need for OS kernel modifications.

Given this, we present **Calico**, a DBMS-managed buffer pool based on *array translation*: each logical page ID serves as an offset into a translation array that maps to buffer frames. This design avoids hash probing and pointer chasing on the translation path, keeps integration non-invasive via the existing PID-based interface, and decouples logical translation from OS page tables. As a result, Calico can back frame memory with 2MB huge pages for TLB efficiency while preserving fine-grained DBMS-managed eviction and I/O. While array translation for buffer management is not new [17], it has seen very limited research and deployment because large, sparse, hierarchical page identifiers [3, 4, 22, 27, 62] in modern DBMSs made flat arrays too expensive in terms of memory overhead. Calico makes the approach practical and performant through three techniques: **multi-level translation** with path caching for managing sparse hierarchical IDs efficiently (Section 4.1), **hole punching** to reclaim cold translation memory (Section 4.2), and **group prefetch** to hide indirection latency during high-fan-out graph traversal (Section 5). On modern hardware, these mechanisms let array translation match OS-based page table translation performance while substantially outperforming hash-table translation across scans, B-tree lookups, and graph-based vector search.

We implement Calico as a drop-in replacement for PostgreSQL's buffer manager with about 2.6K lines of code changes. Relative to PostgreSQL's hash table buffer manager, Calico delivers up to 3.95× in-memory and 6.57× larger-than-memory HNSW throughput, reduces IVFFlat median latency by up to 27.3%, and accelerates sequential scans and join queries, reaching 1.52–2.99× speedups.

Our contributions are:

- **Micro-architectural analysis of buffer pool translation.** We analyze the shortcomings of existing designs and show how user-space array translation matches the speed of hardware-assisted OS-managed translation (Sections 2 and 3).
- **A practical DBMS-managed array-translation buffer pool.** We show how to decouple logical translation from OS page tables so that the DBMS can combine low-overhead array translation, huge-page-backed frames, and fine-grained eviction/I/O control in single design (Sections 4 and 5).
- **Extensive empirical evaluation.** We demonstrate how our approach incurs lower translation overhead across scans, B-tree lookups, and graph-based vector search compared to state-of-the-art designs, while remaining competitive with hardware-assisted translation. Calico delivers up to 3.9× in-memory and 6.5× larger-than-memory speedup in PostgreSQL for vector search while also improving scan-heavy workloads (Section 6).

2 BACKGROUND AND MOTIVATION

We now characterize the workload requirements that matter in modern workloads and use them to motivate a design-space decomposition of buffer-pool translation.

2.1 Workload and Operational Requirements

Translation sits on the critical path of every page access, but different workloads stress it differently. To evaluate designs systematically, we distill modern buffer pool accesses into four patterns that each exercise a distinct aspect of the translation mechanism.

Sequential scan (SS) streams through long contiguous PID ranges (e.g., heap scans, IVFFlat cluster scans [10, 31]). Because hundreds of consecutive translations occur in a tight loop, SS exposes whether a translation mechanism preserves spatial cache locality.

Point lookup (PL) captures latency-sensitive random access such as a B-tree root-to-leaf descent. Each translation is followed by a data-dependent load on the returned frame, so per-access translation latency directly determines throughput.

Range scan (RS) begins with a short dependent lookup phase (e.g., B-tree descent) and then scans consecutive leaf or posting-list pages. It combines the dependency-chain sensitivity of PL with the locality demands of SS.

Graph traversal (GT) captures irregular, high-fan-out graph traversals such as HNSW beam search [19, 30, 50]. Expanding a single node may issue dozens of neighbor translations simultaneously, so GT rewards designs that expose memory-level parallelism and penalizes those with serializing dependency chains.

These four patterns span the locality and parallelism axes that modern buffer pools must handle simultaneously. We label them SS, RS, PL, and GT throughout the paper and use them in Table 1 and Section 3 to expose the trade-offs of each design.

2.2 Design Space of Buffer Pools

Buffer pool designs can be roughly classified along two orthogonal axes: the **translation data structure** and **where the translation state is stored**. Table 1 maps this design space by using these base translation mechanisms as rows. The columns evaluate each design across key system properties and workload behaviors, capturing both their baseline performance and their compatibility with supplementary optimizations (*Predictive Translation*, *Pointer Swizzling*).

DBMS-managed hash-table pools: Hash table translation is the most widely used design in production systems [5, 57, 59, 66, 74] where translation state is managed by the DBMS. A hash table maps page identifiers (PIDs) to buffer frames, giving the DBMS control over eviction policies and I/O scheduling. The trade-off is critical-path overhead: collision resolution, synchronization, and pointer

Table 1: Key Properties and Workload Efficiency of Buffer-Pool Designs – Comparison of the three translation-data-structure classes. Workload-efficiency cells report SS, RS, PL, and GT using color-coded badges, where green, yellow, and red indicate strong, mixed, and weak efficiency. GT† in the pointer-swizzling column denotes limited support for graph-style workloads due to multi-parent references. The property columns summarize huge-page friendliness, required OS modifications, I/O control, and memory-overhead scaling.

Translation Structure	Where Stored	Key Properties				Translation Performance		
		4KB I/O with Huge-page-backed Frames	OS mods	I/O control	Memory overhead	Baseline	+ Predictive Translation	+ Pointer Swizzling
Hash Table Translation	DBMS	yes	none	full	*O*(# cached pages)	SS RS PL GT	SS RS PL GT	SS RS PL GT†
Hardware Radix Page Table	OS Kernel	no	mixed	mixed	*O*(# storage pages)	SS RS PL GT	\	\
Array Translation (Calico)	DBMS	yes	none	full	*O*(# storage pages)	SS RS PL GT	SS RS PL GT	SS RS PL GT†

chasing create dependency chains that limit memory-level parallelism for high-fanout workloads. Hashing also scatters consecutive PIDs, destroying spatial locality on scans; under high thread counts, lock/atomic synchronization increases coherence traffic.

OS-managed page-table translation: This approach stores translation state in a radix-tree page table managed by the OS and MMU hardware [40, 58, 76, 90]. *File-backed* `mmap` delegates page faults, caching, and eviction to the OS [26, 70, 74], reducing DBMS control over replacement and I/O scheduling. *Virtual-memory-based* systems such as vmcache keep translation state in OS page tables but move policy into the DBMS. In **vmcache** (without exmap), the DBMS performs explicit I/O and eviction using standard kernel interfaces, which preserves user-space deployability but pays higher per-page VM-management overhead such as TLB-shootdown, especially on high-end storage devices. Kernel changes are required to improve scalability of page fault handling and eviction [40, 90], but such modifications face deployment barriers.

Page granularity mismatch: OS-managed page-table translation fundamentally shares a limitation. DBMSs need fine-grained page management and I/O (typically 4–32KB), while huge pages (2MB) are important for TLB efficiency. When translation state resides in hardware page tables, the OS cannot cleanly evict a single 4KB subpage from a mapped 2MB huge page; it must evict the full 2MB region [33], split the huge page [51, 54], or fail/refuse the operation [2, 6]. Hence these systems must choose between 4KB pages (fine-grained eviction and I/O, higher TLB pressure) and 2MB pages (lower TLB pressure, high I/O amplification). For this reason, vmcache chooses 4KB pages in favor of fine-grained I/O [40].

Pointer swizzling and predictive translation: Conceptually, they can be applied to any mechanism that exposes a modifiable translation path, but in practice they are most effective for DBMS-managed translation structures; for OS-managed page-table translation, the structure is fixed in hardware/OS page tables. Pointer swizzling [20, 39, 42, 58, 83] reduces translation overhead by replacing logical IDs with pointers, but introduces invasive coupling. Evictions require unswizzling/validation bookkeeping, which is hard for multi-parent references (e.g., graph fan-in, sibling links, secondary-index backlinks). LIPAH [63] improves coverage but still requires in-page hints and validation logic. Predictive translation [93] keeps a hash-table translation layer but deterministically assigns storage pages to preferred frame positions so the CPU can speculatively access the likely frame while translation is in progress. It is effective when hot pages remain in preferred positions, but benefits are dependent on workload and buffer-pool size.

Array Translation: Array translation uses the page ID directly as an index into a contiguous array of frame references, replacing hash computation and probing with a single indexed memory load. Despite being discussed historically [17], it has seen limited research and deployment because large, sparse page-identifier spaces make a naive flat array impractical. Calico targets this underexplored DBMS-managed design point. The next section performs a microarchitectural analysis to establish whether array translation can match hardware-assisted alternatives.

3 TRANSLATION PERFORMANCE ANALYSIS

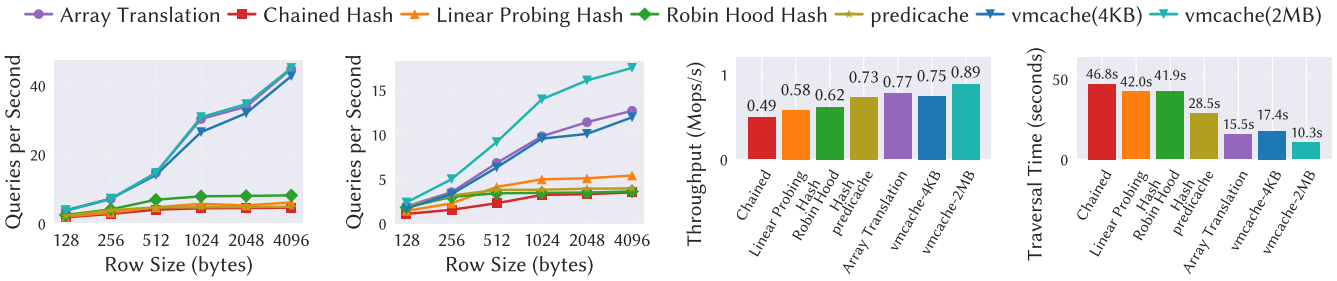
Before addressing sparsity and metadata management in Section 4, we first ask whether array-based translation is fast enough to serve as the foundation of a modern buffer pool. To isolate translation cost, we compare four approaches from Section 2: hash tables, predicache [93], vmcache [40] (4KB and 2MB pages), and Calico's array-based translation. All buffer pools implement the same interface: given an 8-byte page ID, return a pointer to the buffer frame.

Setup: We run on a m7a.8xlarge AWS EC2 instance. All experiments are single-threaded with data fully resident, isolating translation overhead from eviction and synchronization. Page IDs are allocated from 0 sequentially. We use three hash-table baselines: Chained Hash uses chaining for collision resolution (`std::unordered_map`), Linear Probing Hash uses SIMD-accelerated probing (`absl::flat_hash_map`), and Robin Hood Hashing (`robin_hood::unordered_flat_map`). We use OS huge pages to back buffer frames for user-space-managed buffer pools.

We evaluate three workload families that stress different aspects of translation performance: **(1) Scan workloads** (Section 3.1) measure how well each mechanism preserves spatial locality for sequential and random page access patterns in analytical queries and partition-based vector search. **(2) B-tree lookup** (Section 3.2) evaluates translation overhead for OLTP workloads with dependent memory accesses that limit parallelism. **(3) Graph traversal** (Section 3.3) stresses memory-level parallelism with parallel random accesses representative of proximity-graph-based vector search.

3.1 Scans

We measure spatial locality preservation using a scan query on heap pages with row-oriented format (Figures 1a and 1b, Table 2). Sequential scans are common in analytical queries and IVF-based vector search, while random scans occur during index-organized table access. We execute an aggregation query over an integer column and scanning 5GB of data in total. *Sequential* scans access



(a) Sequential scan (heap scan) **(b)** Random range scan (B-tree scan) **(c)** B-tree random point lookup **(d)** Graph BFS traversal

Figure 1: Impact of Translation Across Workloads – (a) Sequential scan: Even the strongest hash baseline is 3.9× slower than Array Translation, which remains close to vmcache (2MB). (b) Random range scan: Array Translation is 2.0× faster than the strongest hash baseline and slightly exceeds vmcache (4KB). (c) B-tree lookup: Hash tables retain clear translation overhead; predicache narrows the gap through speculation, while Array Translation remains faster. (d) Graph BFS: Hash baselines require 41.9–46.8 seconds; predicache helps partially, while Array Translation outperforms vmcache (4KB).

Table 2: Microarchitecture Metrics – Performance counters per page scanned (8KB page, 1024-byte rows) for sum query on 50GB buffer pool scanning 5GB data. Sequential scan: consecutive page IDs (e.g., heap scan); Random scan: non-consecutive page IDs (e.g., B-tree leaf scan). Metrics averaged across all pages scanned.

Config	QPS	IPC	Inst	LLC Ref	LLC Miss	DTLB Miss	Cycles
Sequential Scan (heap scan)							
Chained Hash	4.38	0.15	115	21.2	3.09	1.11	755
Linear Probing Hash	5.67	0.21	122	18.9	2.53	0.81	585
Robin Hood Hash	7.85	0.31	115	17.9	2.79	0.93	373
predicache	4.67	0.17	118	23.5	7.55	0.54	702
Array Translation	30.3	0.77	83.8	13.8	1.33	0.002	109
vmcache (4KB)	26.4	0.64	80.1	14.2	1.75	1.25	125
vmcache (2MB)	30.8	0.74	80.1	14.0	1.42	0.002	108
Random Scan (B-tree leaf scan)							
Chained Hash	3.20	0.15	155	34.0	10.9	2.35	1033
Linear Probing Hash	4.95	0.24	161	29.1	8.09	1.58	669
Robin Hood Hash	3.43	0.14	115	24.5	7.44	1.53	849
predicache	3.78	0.14	119	27.4	9.43	1.01	870
Array Translation	9.78	0.36	123	22.5	5.22	0.56	338
vmcache (4KB)	9.52	0.34	120	20.8	7.36	2.28	347
vmcache (2MB)	13.9	0.50	120	20.7	5.01	0.56	237

consecutive page IDs (simulating heap scans), while *random* scans access non-consecutive page IDs (simulating B-tree leaf scans). This workload stresses whether translation preserves adjacency and lets the CPU prefetch scan streams. Figures 1a and 1b show that Array Translation reaches 30.3 QPS on sequential scan and 9.78 QPS on random scan. The strongest hash baseline reaches 7.85 QPS (Robin Hood Hash) and 4.95 QPS (Linear Probing Hash), respectively, leaving 3.9× and 2.0× gaps. Array Translation remains close to vmcache (2MB) at 30.8 QPS for sequential scan and slightly exceeds vmcache (4KB) at 9.52 QPS for random scan. The counters in Table 2 help explain these gaps. On sequential scan, Array Translation executes 83.8 instructions and incurs 1.33 LLC misses per page, compared with 115 instructions and 2.79 LLC misses for the strongest hash baseline. On random scan, it requires 338 cycles and 5.22 LLC misses per page, compared with 669 cycles and 8.09 LLC misses for the strongest hash baseline. Contiguous translation entries therefore preserve locality across sequential page IDs, while the array lookup also shortens the translation path when page IDs are non-consecutive.

Table 3: Microarchitecture metrics per B-tree lookup for YCSB-C workload on 24GB buffer pool with 100M records (128 bytes each) – Single-threaded read-only point queries with a B-tree depth of 4. Metrics averaged across all lookups.

Config	Mops/s	IPC	Inst	LLC Ref	LLC Miss	DTLB Miss	Cycles
Chained Hash	0.49	0.30	1294	44.5	22.7	3.69	4,251
Linear Probing Hash	0.58	0.37	1327	39.3	17.8	2.18	3,583
Robin Hood Hash	0.62	0.40	1248	48.2	20.7	3.31	3,105
predicache	0.73	0.45	1285	28.9	16.0	1.34	2,863
Array Translation	0.77	0.42	1143	20.7	12.2	0.46	2,707
vmcache (4KB)	0.75	0.40	1124	24.3	14.3	2.48	2,797
vmcache (2MB)	0.89	0.48	1124	20.1	12.2	0.47	2,347

3.2 Point Lookup

B-tree point queries represent the core OLTP workload. Each lookup traverses from root to leaf, accessing random pages with dependent loads that limit memory-level parallelism. We use YCSB-C (read-only) on 100M records (128 bytes each, 24GB buffer pool, 4KB pages) to evaluate translation overhead in this common access pattern with results in Figure 1c and Table 3. The hash baselines span 0.49–0.62 Mops/s, with Robin Hood Hash at the upper end. predicache reaches 0.73 Mops/s by overlapping part of hash-based translation with frame access through CPU speculative execution. Array Translation reaches 0.77 Mops/s, 25% faster than the strongest hash baseline and close to vmcache (4KB). Relative to Robin Hood Hash, it executes 8% fewer instructions, incurs 41% fewer LLC misses, and requires 13% fewer cycles per lookup. These reductions follow from its compact, array translation path.

3.3 Graph Traversal

Graph BFS traversal exposes translation's impact on memory-level parallelism, which is an important property for modern vector search workloads. We perform breadth-first traversal on a 5M-node graph (simulating HNSW vector index beam search) where each node connects to 44 random neighbors (4KB page per node). When visiting a node, the algorithm probes all neighbors. The workload is deliberately compute-light to maximize buffer pool stress and expose translation bottlenecks. The results are shown in Figure 1d and Table 4. This workload stresses whether translation preserves high memory-level parallelism when each node expands many neighbors. The hash baselines require 41.9–46.8 seconds and

Table 4: Microarchitecture metrics per graph node visited (including neighbor probes) for BFS traversal on 5M-node proximity graph – Single-threaded traversal from same starting node. Metrics averaged across all graph nodes visited during full graph traversal.

Config	Time (s)	IPC	Inst	LLC Ref	LLC Miss	DTLB Miss	Cycles
Chained Hash	46.8	0.18	3552	548	337	110	20,206
Linear Probing Hash	42.0	0.22	3937	502	255	91	18,107
Robin Hood Hash	41.9	0.25	4012	402	225	81.8	16,071
predicache	28.5	0.36	4385	307	183	51	12,335
Array Translation	15.5	0.63	2248	265	142	60	6,671
vmcache (4KB)	17.4	0.28	2082	261	147	39	7,476
vmcache (2MB)	10.3	0.48	2079	175	114	27	4,367

predicache reduces traversal time to 28.5 seconds. Array Translation reaches 15.5 seconds, a 2.7×/1.8× improvement over the strongest hash baseline/predicache, while also outperforming vmcache (4KB). Relative to Robin Hood Hash, Array Translation executes 44% fewer instructions, incurs 37% fewer LLC misses, and requires 58% fewer cycles per node. The larger cycle reduction reflects independent array loads that expose parallel neighbor translations without hash-probe dependencies, preserving high memory-level parallelism.

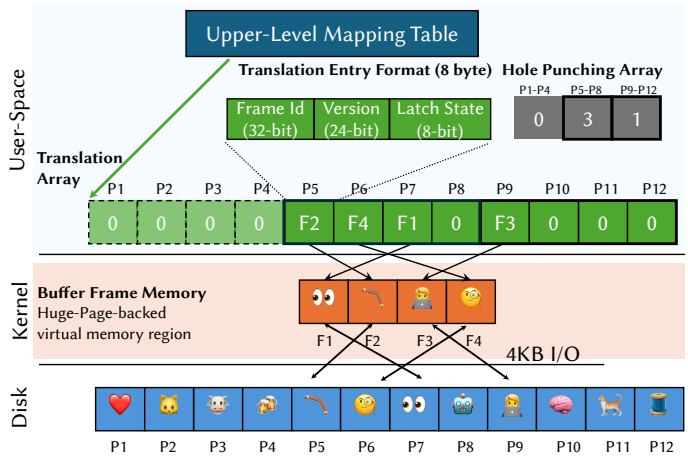


Figure 2: Calico Buffer Manager Architecture – Pages P5–P7 and P9 are resident in frames F1–F4; all other entries are zero (evicted). The upper-level mapping table resolves prefixes to per-region translation arrays whose 64-bit entries encode frame ID, version, and latch state. Frame memory is huge-page-backed; the DBMS performs 4KB I/O independently. The Hole Punching Array tracks one count per entry group (P1–P4: 0, reclaimed; P5–P8: 3; P9–P12: 1). Groups are shown as 4 entries for readability; in practice each spans one 4KB OS page (512 entries).

4 Calico DESIGN

Although our evaluation in Section 3 demonstrated that array translation outperforms hash-table and matches OS-page-table alternatives across sequential, random, and graph-traversal access patterns, a naive flat array is impractical for production DBMSs with large and sparse page-identifier space that exceed what a contiguous array can cover efficiently. This section presents the design of Calico, which makes array translation viable.

Design goals: Translating the raw performance advantage of arrays into a deployable buffer-pool design requires satisfying four

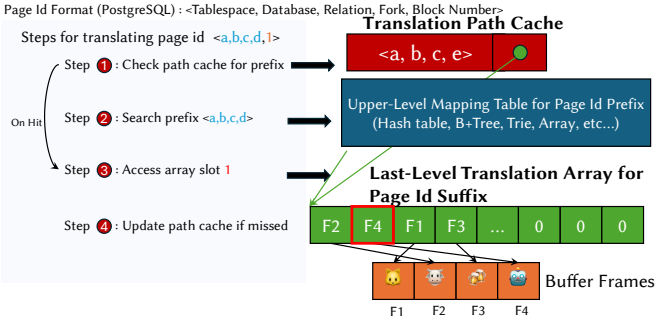


Figure 3: Hierarchical Translation with Path Caching – Calico decomposes each page identifier into a *prefix* and a *suffix*. The figure walks through four steps: (1) check the translation path cache; (2) on a miss, resolve the prefix through an upper-level index (e.g., radix tree, hash table, B$^+$-tree, or trie) to obtain a last-level translation array; (3) use the suffix to directly index that array on the hot path; and (4) update the path cache with the resolved prefix-to-array mapping. Unlike hash-table translation, which hashes the entire PID as an opaque input, Calico exploits PID locality in database systems by decomposing it into a prefix and suffix, then directly indexing the last-level translation array with the suffix without probing or equality checks.

goals simultaneously: ❶ **Sparse PID support** – handle large and sparse page-identifier spaces without allocating a monolithic flat array spanning the entire logical domain; ❷ **Array-fast-path preservation** – retain single-load array translation on the hot path so that the performance gains established in Section 3 carry over to the full system; ❸ **Fine-grained DBMS control with huge-page efficiency** – keep eviction, replacement, and I/O scheduling under DBMS control at page granularity while backing frame memory with huge pages for TLB efficiency; ❹ **Active memory reclamation** – reclaim physical memory from cold translation regions so that the translation memory overhead tracks only the working set.

Overview. Calico separates DBMS-managed logical translation from OS-managed physical frame backing (Figure 2). To handle sparse PID spaces (❶), Calico organizes translation hierarchically, allocating per-region last-level arrays only for active regions; because modern DBMS workloads exhibit strong locality within relations and indexes, a lightweight path cache ensures that the common case reduces to a single array translation (❷). Frame memory is huge-page-backed and managed independently of translation arrays, so page-granularity eviction does not require OS page-table modifications (❸). An active hole-punching mechanism tracks and reclaims translation memory from fully cold regions (❹). The remainder of this section details hierarchical translation and path caching (Section 4.1), on-demand array memory management (Section 4.2), and the buffer-pool algorithms (Section 4.3).

4.1 Hierarchical Translation and Path Caching

To tackle the problem of large sparse PID spaces, Calico introduces hierarchical translation so memory is proportional to active regions rather than the full logical PID domain. Upper levels map PID prefixes to lower-level translation arrays, and those lower-level arrays are allocated only when a prefix becomes active. Prefixes with no resident pages keep no materialized last-level array, so empty PID regions do not consume translation memory.

While such multi-level translation introduces extra indirection, the key observation is that modern DBMSs organize page identifiers hierarchically, which induces strong locality in prefix components. B-tree traversals, HNSW traversals, and scans repeatedly access pages within the same relation/index region [55, 60, 61, 65, 67, 68]. Calico exploits this locality using translation-path caching. Figure 3 walks through translation of page identifier $p = (prefix, suffix)$: first check the path cache, then resolve the prefix through the upper-level index on a miss, then perform suffix-based array translation in the resolved last-level array, and finally update the cache with the resolved prefix-to-array mapping. When subsequent accesses reuse the same prefix, upper-level traversal is bypassed and only the final array lookup remains on the hot path. In practice, because successive accesses within a B-tree traversal, HNSW search, or scan almost always share the same prefix, a single thread-local cache entry storing the last resolved (prefix, last-level array) pair suffices to achieve good hit rates.

The remaining design question is how to choose the prefix/suffix split. A natural decomposition follows the storage hierarchy already present in most DBMSs: the *prefix* identifies a stable container region (e.g., database/table/index or relation/fork), and the *suffix* identifies the page within that region. In most DBMS layouts, the suffix is the page or block number within a table/index, which keeps the last-level translation array densely indexable by array offset. Take PostgreSQL as a concrete instantiation. For PID <Tablespace, Database, Relation, Fork, Block>, Calico uses prefix = <Tablespace, Database, Relation, Fork> to select the last-level translation array and suffix = <Block Number> for direct indexing in that array (Figure 3). This matches observed locality where accesses repeatedly remain in the same relation/fork region while block numbers vary. The same principle generalizes to MySQL's hierarchical <space_id, page_no> identifiers. Additional levels can be added when PID-space sparsity requires them.

Discussion. Array translation can also be viewed as a specialized hash table with an identity hash function and one large bucket per possible page identifier. Unlike a conventional hash table, the bucket is sized for its dense, bounded key domain rather than for the number of resident pages, so keys are implicit in the bucket index and collision handling is unnecessary. The mapping therefore retains the user-space residency state and eviction control of a hash-table buffer manager while removing hash computation, stored keys, equality checks, and probing from the translation path. It also avoids the randomized bucket access that a conventional hash table would introduce when translating a sequence of consecutive page identifiers, so sequential page accesses remain spatially local in the translation structure.

4.2 On-demand Array Memory Management

After upper-level prefix resolution, every access goes through a last-level translation array. This hot path must satisfy three requirements: (1) fast array-indexed translation, (2) concurrency control for reads/updates/eviction, and (3) memory usage proportional to the working set despite sparse logical PID spaces.

Calico encodes each `TranslationEntry` as one 64-bit atomic word. The **latch state** (8 bits) supports exclusive/shared synchronization, the **version number** (24 bits) enables optimistic validation for lock-free reads on the buffer frame, and the **frame ID** (32 bits) identifies the buffer frame. A single load returns both translation and concurrency metadata. Translation is as follows:

```
1 TranslationEntry* te = &TranslationArray[pageId];
2 Frame* frame = frameMem + te->getFrameId();
```

Memory efficiency and reclamation: The flat translation array introduces a potential space overhead: for a 16TB SSD with 4KB pages, the table requires 32GB of memory (4G entries × 8 bytes). Calico leverages the OS's on-demand memory allocation mechanism combined with **hole-punching** to reduce the *physical* memory usage so that it is proportional to the working set size.

Zero-Page Copy-on-Write at Startup: The translation array is allocated via `mmap`. Windows provides analogous virtual-memory reservation APIs [56]. The reserved region is initially backed by a shared zero-filled OS physical page [48]. When a thread first reads an entry, the OS triggers a page fault and establishes a read-only mapping to the shared zero-filled physical page, setting the copy-on-write (COW) bit in the page table entry. Only when an entry is updated (e.g., in `calico_page_fault_handler`) does the OS allocate a physical page, copy the zero page content, and update the mapping. This on-demand allocation pattern aligns with Calico's access pattern: entries for cold pages that are never loaded remain on the shared zero page indefinitely, consuming no physical memory. Thus, the virtual size of the translation array can be proportional to the logical page ID space, while its physical memory footprint tracks only the working set.

Zero-Value Entry Design for Lazy Loading: To exploit this mechanism, Calico's `TranslationEntry` encoding is carefully designed such that an all-zero 64-bit value represents an *evicted* page state. Specifically, when all 64 bits are zero: (1) the frame ID field is 0, interpreted as `INVALID_FRAME`, indicating no buffer frame is allocated; (2) the latch state field is 0, representing `Evicted`; and (3) the version number is 0. This invariant ensures that at system startup, when the entire translation array is logically filled with zeros (via the shared zero page), every entry correctly represents an evicted page requiring a page fault on first access. Crucially, when a page is evicted (`calico_evict_victim`), Calico writes a zero value back to the entry, returning it to the invalid state. This allows further memory reclamation through active hole-punching.

Hole-Punching Array: While lazy allocation via OS demand paging ensures translation arrays consume physical memory proportional to active entries, cold regions may remain physically allocated long after pages are evicted. To address this, Calico introduces the Hole-Punching Array (`HPArray`), a lightweight reference counting structure that tracks the number of valid (non-evicted) translation entries within each OS page of the translation array, enabling active memory reclamation.

The `HPArray` is a flat array of atomic 32-bit counters, where each counter tracks one **entry group**: consecutive translation entries fitting within a single OS page. For a translation array with N entries, `HPArray` requires $\lceil N/\text{entries\_per\_OS\_page} \rceil$ counters. With 512M translation entries and 4KB pages, this requires 1M counters (4MB); with 2MB huge pages, only 2048 counters. The `HPArray`

itself is allocated lazily via `mmap`, consuming physical memory only on first write. Each counter reserves one bit as a lock to coordinate hole-punching operations (Section 4.3).

Memory Overhead Discussion: Calico's translation array stores one 8-byte entry per 4KB storage page, so its worst-case overhead is proportional to on-storage data size. However, this is less than vmcache [40], which requires ≈16 bytes per 4 KB storage page: 8 bytes of x86-64 page-table entries (unavoidable for any `mmap`-based design) plus an 8-byte page-state. For a 1 TB SSD, Calico's worst-case translation footprint is 2 GB versus vmcache's ≈4 GB. As flash is roughly 50× cheaper per byte than DRAM [40], Calico's overhead adds ≈10% to the flash price.

In practice, Calico is competitive in terms of memory overhead. Real-world DBMS workloads exhibit spatial locality [9], so hot pages cluster into contiguous regions and cold regions form large hole-punchable groups. OS page tables retain non-zero swap entries for evicted pages, preventing kernel reclamation; Calico's eviction explicitly zeros entries, enabling hole punching to reclaim cold translation memory. Compared with hash tables, the constant-factor trade-off favors Calico on the hot path. While a hash table stores translation entries only for cached pages, it stores both key and value per entry plus load-factor headroom. For example, predicache [93] uses 40-byte entries at a 50% load factor. Calico's 8-byte translation entries are 5–10× denser, directly reducing cache misses during translation (Section 3).

4.3 Buffer-Pool Algorithms

Algorithm 1 presents Calico's core buffer-pool algorithms for pinning, unpinning, page fault handling, and eviction. We assume a thread-local path cache that stores the most recent `(prefix, lastLevelArray)` mapping for each thread.

Exclusive Pin (Algorithm 1): To modify a page, a thread must acquire an exclusive lock on the translation entry. To do this, it atomically loads the entry, checks if the page is resident (valid frame ID), and uses compare-and-swap (CAS) to transition from `Unlocked` to `Locked`. If the page is not resident (`INVALID_FRAME`), it triggers the page fault handler. On success, the thread receives a pointer to the frame holding the page data. Note that shared pins can be implemented similarly by storing the number of readers in the latch state of `TranslationEntry`.

Optimistic Read (Algorithm 1): For read-heavy operations, Calico exposes a lock-free optimistic read interface. The reader snapshots a `TranslationEntry`, executes the read function on the target frame, and then validates that the entry version and frame ID are unchanged and that the entry is not locked. This avoids atomic pin/unpin traffic on the read path while preserving correctness under concurrent eviction and modifications.

Exclusive Unpin (Algorithm 1): Releasing an exclusive pin is a single atomic operation: set the lock state to `Unlocked` and increment the version number. The version bump invalidates any concurrent optimistic readers that observed the old version.

Page Faulting (Algorithm 2): When a page is not resident, the fault handler acquires an exclusive lock on the translation entry and double-checks the frame ID (another thread may have already loaded the page). If still invalid, it allocates a frame (from the free list or via eviction), issues I/O to load the page data, then atomically increments the reference count in the `HPArray` for the OS page

Algorithm 1: Calico Buffer Pool Algorithms

```
1  Function GET_TRANSLATION_ENTRY(pageId):
2      (prefix, suffix) ← split_pid(pageId)
3      if TLSCache.prefix = prefix then
4          last_level_array ← TLSCache.lastLevelArray
5      else
6          last_level_array ← lookup_leaf_table(prefix)
7          TLSCache ← (prefix, last_level_array)
8      return &last_level_array[suffix]
9  Function PIN_EXCLUSIVE(pageId):
10     while true do
11         te ← Get_Translation_Entry(pageId)
12         old_e ← *te // Atomic load
13         if old_e.frameId = INVALID_FRAME then
14             page_fault_handler(pageId, te)
15             continue
16         if old_e.state = UNLOCKED and
             te.CAS(old_e, (old_e.frameId, old_e.version, LOCKED)) )
             then
17             return frameMem + old_entry.frameId
18 Function UNPIN_EXCLUSIVE(pageId):
19     Get_Translation_Entry(pageId).bump_version_unlock()
20 Function OPTIMISTIC_READ(pageId, read_func):
21     while true do
22         te ← Get_Translation_Entry(pageId)
23         old_entry ← *te // Atomic load
24         if old_entry.frameId = INVALID_FRAME then
25             page_fault_handler(pageId, te)
26             continue
27         if old_entry.state = LOCKED then
28             continue // Spin until unlocked
29         read_func(frameMem + old_entry.frameId)
30         new_entry ← *te
31         if old_entry.version = new_entry.version and
             old_entry.frameId = new_entry.frameId then
32             return // Success
```

Algorithm 2: Calico Page Fault Handler

```
1  Function PAGE_FAULT_HANDLER(pageId, te):
2      while ¬te.try_lock() do
3          continue
4      if te.frameId ≠ INVALID_FRAME then
5          te.unlock()
6          return
7      frameId ← allocate_frame_or_evict()
8      io_read_page(pageId, frameMem + frameId)
       // Atomically increment counter for this entry group
9      groupIdx ← group_index(te)
10     increment_hparray(groupIdx)
11     te.set_frame_and_unlock(frameId)
```

Algorithm 3: CALICO Eviction with Hole-Punching

```
1  Function EVICT_VICTIM():
2      victimId ← select_victim_page() // CLOCK, LRU, etc.
3      te ← Get_Translation_Entry(victimId)
4      while ¬te.try_lock() do
5          continue
6      frameId ← te.frameId
7      write_back_if_dirty(frameMem + frameId)
8      te.frameId ← INVALID_FRAME // Zero out frame id
       // Atomically decrement HPArray refcount
9      groupIdx ← group_index(te)
10     count ← HPArray(victimId)[groupIdx].LOCK_AND_DEC()
11     te.unlock_evicted() // Now zero out latch state
12     if count = 0 then
           // Last valid entry evicted, reclaim memory
13         madvise(group_base_addr(te), PG_SIZE, MADV_DONTNEED)
14     HPArray(victimId)[groupIdx].UNLOCK()
15     return frameId
```

group containing this translation entry. Finally, it updates the entry with the new frame ID and releases the lock. The translation entry lock prevents duplicate I/O when multiple threads fault on the same page, while incrementing the counter before publishing the frame ID ensures the group cannot be hole-punched during page fault.

Eviction (Algorithm 3): CALICO uses a standard replacement policy (CLOCK) to select a victim page. The algorithm acquires an exclusive lock on the victim's translation entry, writes back the frame if dirty, and invalidates the entry by setting the frame ID to `INVALID_FRAME`. Next, it atomically locks the metadata counter and decrements the reference count for the corresponding OS page group. Crucially, the translation entry is unlocked only *after* acquiring the metadata lock. If the count reaches zero (all entries in the group evicted) after decrement, the thread performs hole-punching via passing `MADV_DONTNEED` hint to OS while still holding the metadata lock, then releases the lock. This ordering ensures that concurrent page faults must wait for the metadata lock before incrementing the counter, preventing any thread from installing a new frame ID in a page being hole-punched. Unlike vmcache, which invokes `madvise` on every data-page eviction, CALICO's hole punching fires only when an entire OS page worth of translation entries becomes cold, making it far less frequent.

5 IMPLEMENTATION AND OPTIMIZATIONS

This section presents CALICO's group prefetch interface and then describes system integration in PostgreSQL and `pgvector`.

5.1 Group prefetch

Modern workloads like graph-based vector search exhibit predictable access patterns: future page accesses are often known in advance. In proximity-graph-based vector search [19, 30, 50], when visiting a graph node, the algorithm must probe neighboring nodes whose IDs are stored in the current node. CALICO exploits this predictability through a **group prefetch interface** that issues parallel reads to the translation array and prefetches buffer frame memory. This parallelism operates at two levels:

- **Memory-Level Parallelism**: For resident pages, parallel translation array reads allow the CPU to issue multiple independent loads simultaneously. Modern CPUs can issue parallel memory loads [43, 44, 49], hiding memory latency.
- **I/O-Level Parallelism**: For non-resident pages, knowing multiple page IDs in advance enables batched asynchronous I/O, saturating storage bandwidth and reducing total latency.

Algorithm 4 presents the group prefetch algorithm. The interface accepts page IDs with corresponding in-page offsets for targeted prefetching. The algorithm operates in three phases: (1) issues prefetch instructions for translation entries to bring them into cache; (2) issues prefetch instructions for resident page frames at specified offsets while collecting non-resident page IDs; (3) submits batched reads for non-resident pages. This exposes batching semantics, enabling indexes to exploit modern hardware capabilities.

5.2 PostgreSQL Integration

We implement multi-level CALICO in PostgreSQL v18 [1], which uses a 5-level hierarchical page identifier BufferTag [27]. We use the last-level 32-bit BlockNumber as the suffix and the remaining fields as the prefix identifying the relation. Our implementation uses a hash table for top-level mapping and flat arrays for last-level translation arrays. We extended PostgreSQL's buffer manager with CALICO's group prefetch and optimistic read interfaces. The implementation adds approximately 2.6K lines of C code. The modifications are mostly isolated within the buffer manager, preserving compatibility with existing components.

Translation Path Caching via Thread-Local Storage. As described in Section 4.1, each worker thread caches the most recently resolved (prefix, last-level array) mapping in `thread_local` storage. In PostgreSQL, the prefix corresponds to the upper fields of the `BufferTag`; on each buffer manager call, the thread compares the current tag's prefix bits against the cached value and, on a hit, skips the top-level hash lookup entirely.

pgvector Integration. We then modified the `pgvector` extension[2] (about 600 lines) to leverage these interfaces during HNSW graph traversal. PostgreSQL's existing page access requires complex pin/unpin operations using atomic writes that limit memory-level parallelism [13, 52]. By using CALICO's optimistic read interface, pgvector bypasses these atomic operations. When visiting a graph node, pgvector extracts neighbor page IDs and prefetch neighbors, then CALICO uses `calico_optimistic_read` to access neighbor data without pin/unpin overhead. When validation fails, we revert to standard pin/unpin operations to avoid repeated wasted work.

6 EXPERIMENTAL EVALUATION

Our evaluation validates CALICO's performance and scalability across modern workloads. We aim to evaluate:

(1) Array-based translation overhead compared to mmap, pointer swizzling, hash tables on vector search and OLTP workloads.
(2) End-to-end performance gains in PostgreSQL.
(3) Memory overhead of array-based translation with hole-punching.

[1]The modified PostgreSQL is available at https://github.com/zxjcarrot/postgres_calico/tree/rel_18.
[2]The modified `pgvector` extension is available at https://github.com/zxjcarrot/pgvector.

Algorithm 4: CALICO Group Prefetch Algorithm

```
Function GROUP_PREFETCH(pids, offsets):
    non_resident_pids ← [ ]
    // Prefetch translation entries
    foreach pageId ∈ pids do
        te ← Get_Translation_Entry(pageId)
        prefetch(te)
    // Prefetch resident pages and collect non-resident ones
    foreach i ∈ [0, length(pids)) do
        pageId ← pids[i]
        offset ← offsets[i]
        te ← Get_Translation_Entry(pageId)
        if te.frameId ≠ INVALID_FRAME then
            prefetch(FrameMemory[te.frameId] + offset)
        else
            non_resident_pids.append(pageId)
    if non_resident_pids ≠ ∅ then
        io_read_pages(non_resident_pids)
```

(4) Generality of CALICO on different CPU platforms.

Unless stated otherwise, all experiments are conducted on a dual-socket server with two AMD EPYC 7513 processors (32 cores per socket, 64 cores/128 threads total, 2.6 GHz base frequency), 504 GB DDR4 memory, and a Samsung PM9A3 3.84TB NVMe SSD (1M random 4KB read IOPS). The system runs CentOS Linux 8.5. We set the CPU governor to performance mode to ensure consistent results. We use Linux transparent huge pages feature via `madvise(MADV_HUGEPAGE)` hint to enable 2MB huge pages to back frame memory and hash table memory, except vmcache which uses 4KB pages due to the granularity problem.

6.1 Workloads, Baselines, and Metrics

We evaluate CALICO on two workloads:

- **Vector Search**: We evaluate vector similarity search using HNSW index and IVF index with pruning [38]. We build index on buffer manager for accessing pages on storage. We evaluate **CALICO**, **vmcache**, **predicache** [93], **USearch** [79] (specialized in-memory HNSW library using `mmap`), and **hash table variants**. The hash-table variants include Chained Hash, Linear Probing Hash, and Robin Hood Hash.
- **OLTP Workloads with B-tree**: We use YCSB-C (read-heavy) and TPC-C (write-heavy) workloads, both operating on B-tree indexes. Baselines include **vmcache** [40] (mmap-based buffer pool), **predicache** [93], **LMDB** [26] v0.9.31 (a mmap-based key-value store), **WiredTiger** v10.0.2 (hash table), and **LeanStore** [42] at commit 629b41a (pointer swizzling). We use the lowest isolation level for LeanStore and WiredTiger and disable write-ahead-logging to focus on buffer management performance.

We use baselines that are user-space deployable and do not require kernel modifications, as our goal is to provide a practical solution that can be adopted by existing systems without OS changes. To isolate the impact of replacement policy on performance, CALICO, vmcache, predicache, and hash-table-based variants are mostly implemented in the same tested with the same CLOCK-based replacement algorithm. Since this work focuses on reducing translation and page-miss handling overhead, we therefore use end-to-end throughput and latency under identical memory constraints as the primary metric.

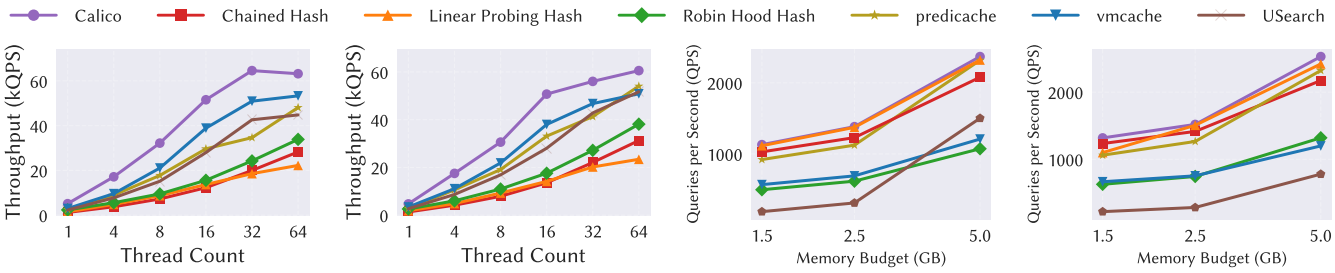
6.2 Vector Search Workloads

We evaluate CALICO on HNSW-based vector search using DEEP [84] (10M vectors, 96D) and SIFT [1] (10M vectors, 128D). We use `caliby` [3], an in-process vector search library implementing HNSW [50], to evaluate different buffer managers. The HNSW index occupies roughly 9.5 GB for DEEP10M and SIFT10M. In-memory runs use a 12 GB buffer pool so the full index remains resident to isolate translation overhead. Larger-than-memory runs reuse the same index with pools of 1.5, 2.5, and 5 GB to isolate I/O and eviction effects. The in-memory experiments vary the thread count from 1 to 64, while the larger-than-memory experiments use 64 threads. All configurations use the same HNSW search parameters (M=16, ef_construction=100, ef_search=50). We enable optimistic reads and group prefetch for all buffer managers. For all hash table-based pools except the lock free hash table, we partition the hash table and use dedicated lock per partition to reduce lock contention when accessing the hash tables. We enable the group prefetch technique for all buffer managers but USearch to utilize I/O parallelism of SSD. USearch relies on kernel paging which does not provide an interface for group prefetching. We turn off OS page cache readahead for Usearch to minimize the I/O amplification as HNSW accesses pages randomly.

In-Memory HNSW. Figures 4a and 4b show throughput scaling when the working set fits entirely in memory with a buffer pool size of 12GB. At one thread, CALICO has the largest single-core advantage: it is 1.72×/1.38× faster than vmcache and 2.23×/1.74× faster than USearch on DEEP10M [84]/SIFT10M [1]. Hash table variants (except predicache) are significantly slower than virtual-memory-based approaches and CALICO due to translation overhead and synchronization costs. Relative to Robin Hood Hash, CALICO delivers 2.18×/1.82× higher throughput at one thread and 1.87×/1.59× at 64 threads on DEEP10M/SIFT10M. While predicache improves over plain hash tables both in terms of performance and scalability via lock-free hash tables and speculative execution, it is still outperformed by CALICO by up to 2× due to mispredictions. At 64 threads, the gap between CALICO and other systems narrows as they approach the memory bandwidth limit of the system.

Larger-than-Memory HNSW. The results in Figures 4c and 4d show that when working sets exceed available memory, CALICO outperforms both vmcache and USearch by 2–6×. The performance gap stems from fundamental differences in I/O management. Relative to Robin Hood Hash, CALICO delivers 1.92–2.28× higher throughput across both datasets and all memory budgets. USearch relies on `mmap` with synchronous page faults: when data is not resident, the OS blocks threads until I/O completes, serializing execution. At 1.5GB memory, USearch only achieves 189-222 QPS (6× worse than CALICO). All other buffer managers (CALICO, vmcache, and hash tables) use prefetch interfaces to issue parallel I/O requests,

[3] The `caliby` source is available at https://github.com/zxjcarrot/caliby.



(a) In-memory DEEP10M **(b)** In-memory SIFT10M **(c)** Larger-than-memory DEEP10M **(d)** Larger-than-memory SIFT10M

Figure 4: HNSW Vector Search: In-Memory and Larger-than-Memory – Left: throughput scaling when data fits in memory. Right: throughput under memory pressure across memory budgets at 64 threads.

avoiding blocking page faults. This explains USearch's steep degradation under memory pressure. Among prefetch-enabled systems, Calico outperforms vmcache by 2× across memory budgets. The root cause for this difference is TLB shootdown overhead. When vmcache evicts pages via `madvise`, the OS interrupts all 64 threads to invalidate TLB entries, which is a synchronization bottleneck that serializes execution. Calico avoids this problem because it evicts pages in user space without OS involvement, eliminating TLB shootdowns for data pages.

IVF Vector Search. We also evaluate a partition-based IVF vector search index [38] on top of buffer managers on four datasets: dbpedia-OpenAI-1M (1536D), GIST1M (960D), MSong (420D), and SIFT10M (128D). IVF vector search probes a set of cluster partitions via sequential scans, stressing spatial locality rather than graph-style random access. We partition each dataset into $\sqrt{N}$ clusters where N is the number of vectors in the dataset and each query probes 16 of them. We use one thread for in-memory experiment to isolate translation overhead without synchronization noise, and 128 threads for larger-than-memory experiment to saturate SSD bandwidth and expose system-level bottlenecks such as eviction and TLB shootdown costs. In-memory IVF runs size the buffer pool to the dataset so the comparison isolates sequential translation, while larger-than-memory runs size the pool to 30% of the dataset to stress I/O and eviction. Figure 5a shows that in-memory at one thread, Calico outperforms vmcache by 1.05–1.19× and predicache by 1.53–1.78× across the four datasets, consistent with the sequential-scan advantage established in Section 3.1. Under memory pressure at 128 threads (Figure 5b), Calico now outperforms vmcache by 1.48–1.85× across all four datasets, because vmcache's per-page eviction triggers frequent cross-core TLB shootdowns that dominate at high thread counts. Against predicache, Calico is similar in throughput as they use the same user-space I/O subsystem.

6.3 OLTP Workloads

We evaluate Calico on YCSB-C and TPC-C using B-tree index in two scenarios: **in-memory** and **larger-than-memory** OLTP. In-memory pools are sized to keep the active dataset resident (16 GB for 12.8 GB of YCSB-C data and 80 GB for 40 GB of TPC-C data), isolating translation and synchronization. Larger-than-memory pools are fixed at 4 GB and 8 GB, respectively, while the dataset grows so that memory pressure increases across the plotted range.

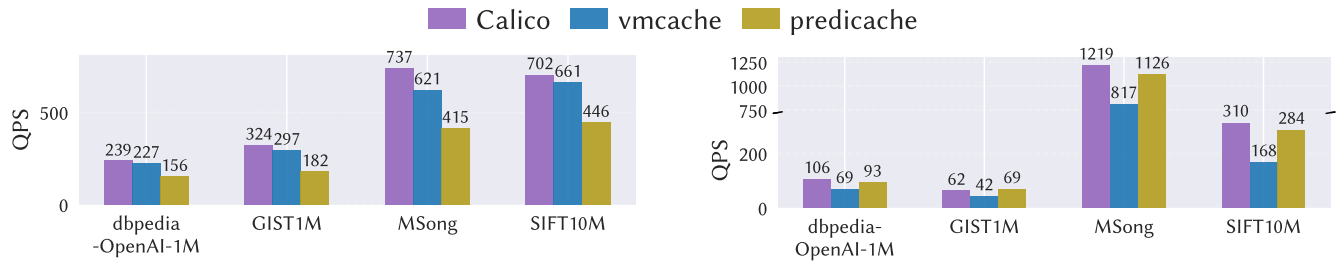
In-Memory. Figure 6 shows throughput scaling from 1 to 128 threads. YCSB-C uses a 16GB buffer pool, while TPC-C uses an 80GB buffer pool. For YCSB-C (Figure 6a), Calico starts slightly behind predicache at low thread counts but scales better, reaching 1.27× vmcache and 1.20× predicache at 128 threads. For TPC-C (Figure 6b), Calico, predicache, vmcache, and LeanStore remain in the same performance tier, with Calico slightly leading at 128 threads. The main takeaway is that Calico matches state-of-the-art OLTP throughput even on write-heavy workloads, without relying on pointer swizzling or prediction. LMDB and WiredTiger remain much slower because their designs do not scale well under heavy write contention. This demonstrates that Calico's array translation is very competitive for OLTP compared to state-of-the-art systems.

Larger-Than-Memory. Figure 7 compares systems as dataset size exceeds buffer pool capacity (4GB for YCSB-C, 8GB for TPC-C) with 64 threads. For YCSB-C (Figure 7a), Calico is competitive at the smallest dataset and then becomes the best system as the index grows. It consistently outperforms vmcache by about 1.6× and remains ahead of predicache at the larger datasets. The key reason is that vmcache is bottlenecked by the kernel under memory pressure as evicted page triggers an expensive TLB-shootdown. Calico avoids that bottleneck by keeping translation and I/O control in user space. WiredTiger remains lower throughput because its 32KB pages amplify I/O cost once the workload spills beyond memory. For TPC-C (Figure 7b), Calico, vmcache, and predicache remain close over the full range, with Calico usually at or near the top. This shows that Calico preserves state-of-the-art OLTP performance not only in memory but also out of memory. LeanStore degrades more noticeably as the dataset grows, while LMDB remains fundamentally constrained by its single-writer design.

6.4 Impact of Calico on PostgreSQL

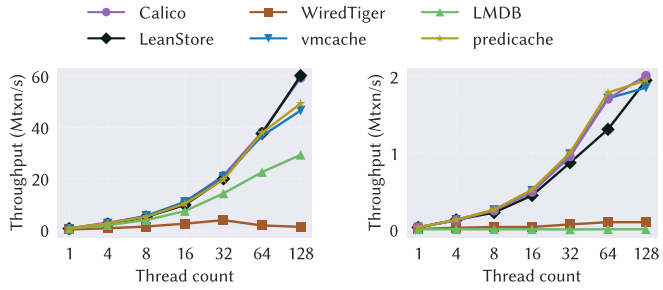
We next evaluate Calico's impact on PostgreSQL v18 across three workload groups: in-memory and larger-than-memory HNSW vector search with `pgvector`, IVFFlat vector search, and OLAP workloads spanning sequential scans, merge joins, hash joins, and nested loop joins. All experiments are driven by a single PostgreSQL client.

Vector Search. Figure 8 reports query throughput on SIFT10M and DEEP10M using `pgvector` with an HNSW index. Starting from the vanilla PostgreSQL buffer manager, we incrementally enable array translation, optimistic reads, and group prefetch. We additionally



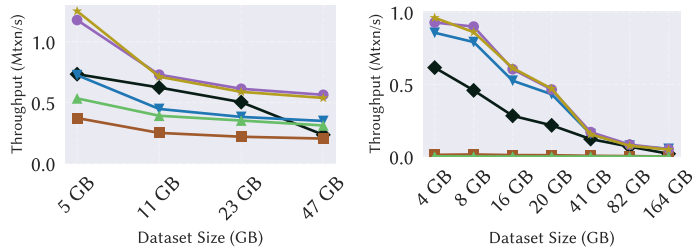
(a) In-memory **(b)** Larger-than-memory

Figure 5: IVF Vector Search Performance – QPS for IVF scan-based vector search on four datasets.



(a) YCSB-C **(b)** TPC-C

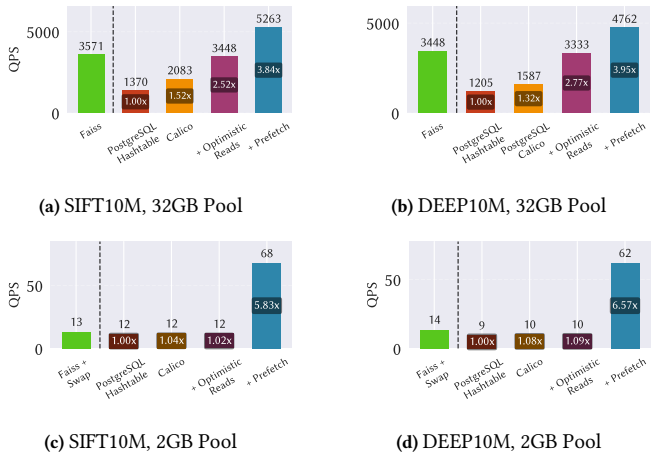
Figure 6: In-memory YCSB-C/TPC-C Throughput Scaling – YCSB-C: 100 M entries=12.8 GB, 16 GB buffer pool; TPC-C: 200 warehouses=40 GB, 80 GB buffer pool



(a) YCSB-C: Throughput **(b)** TPC-C: Throughput

Figure 7: OLTP Workloads (Larger-than-Memory) – Measured throughput for YCSB-C (4 GB buffer pool) and TPC-C (8 GB buffer pool).

include Faiss [16], an in-memory vector search library, as an upper-bound reference under the same parameters. The translation path cache achieves a hit rate close to 100% because PostgreSQL stores all pages of an HNSW index within a single relation. The 32 GB shared-buffer setting keeps the HNSW index resident so the comparison isolates translation and synchronization overhead; the 2 GB setting covers roughly one fifth of the index so that I/O and prefetch dominate. For SIFT10M (Figure 8a), array translation alone yields a 1.52× speedup over the hashtable baseline. Adding optimistic reads raises this to 2.5× by eliminating atomic pin/unpin overhead and cache coherence traffic [13, 52]. Full Calico with group prefetch reaches 3.84×, demonstrating that workload-aware prefetching exploits memory-level parallelism. DEEP10M (Figure 8b) exhibits the same progression: 1.32× with array translation, 2.77× with optimistic reads, and 3.95× with the full configuration.



(a) SIFT10M, 32GB Pool **(b)** DEEP10M, 32GB Pool

(c) SIFT10M, 2GB Pool **(d)** DEEP10M, 2GB Pool

Figure 8: PostgreSQL pgvector HNSW Performance – Bars show query throughput as Calico optimizations are incrementally enabled on top of the PostgreSQL hashtable baseline (ef_search=64). Top row: data fits in memory (32GB pool); bottom row: index exceeds memory (2GB pool). Faiss (in-memory library) is included as an upper-bound reference.

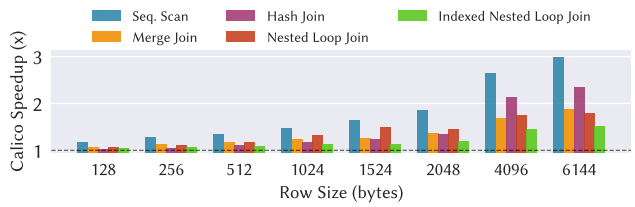


Figure 9: PostgreSQL Workload Speedup Across Row Sizes – Relative speedup of Calico over PostgreSQL Hashtable for sequential scan, merge join, hash join, nested loop, and index nested loop workloads. Each workload is normalized to its own Hashtable baseline of 1.00×; row sizes range from 128 to 6144 bytes.

Larger-than-memory. The bottom row of Figure 8 repeats the comparison with a 2GB buffer pool, in which the HNSW index far exceeds memory. I/O latency dominates in this regime, so array translation and optimistic reads provide only modest gains (1.04–1.09×). Group prefetch, by contrast, hides I/O latency through parallel requests along the graph structure, delivering 5.83× on SIFT10M and 6.57× on DEEP10M. To enforce the same memory budget, we also run Faiss under a 2GB Linux cgroup, forcing it to page through OS virtual memory. Because Faiss paging is synchronous, it cannot

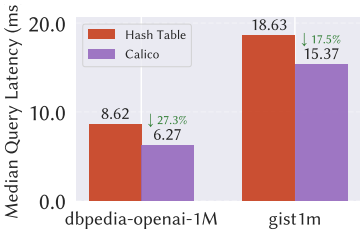


Figure 10: IVFFlat Query Latency in PostgreSQL – Median IVFFlat query latency on DBPedia OpenAI-1M (1536D) and GIST1M (960D) datasets.

exploit Calico's batch group prefetch and is outperformed by up to 5.2×.

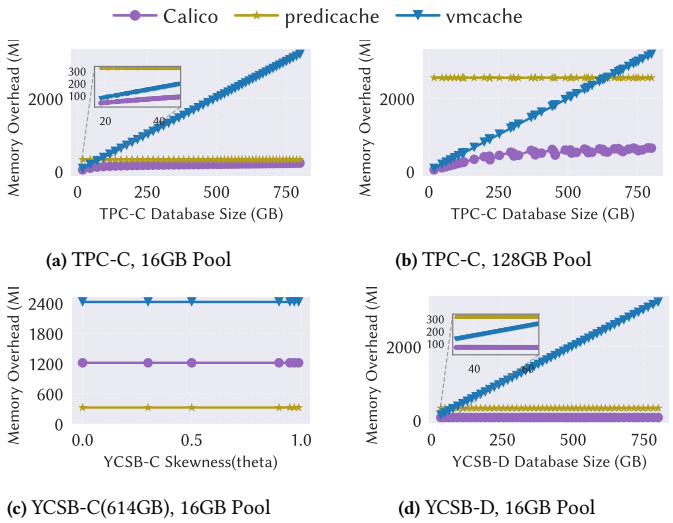
IVFFlat Query. Figure 10 reports IVFFlat query latency on DBPedia OpenAI-1M and GIST1M with a 32GB buffer pool. At identical Recall@3 (0.91 and 0.87), Calico reduces median latency by 27.3% and 17.5%, respectively, and the buffer-manager share of execution time drops from 15–28% to 1–4%.

OLAP Workloads. Figure 9 compares Calico with the PostgreSQL hash table buffer manager on sequential scan and four join queries over row sizes from 128 to 6144 bytes. The 15 GB scan and the join workloads remain resident in a 128 GB shared-buffer setting, isolating buffer-manager overhead. Because their cardinalities differ, each workload is normalized to its own hash table baseline. We observe that all workloads benefit from Calico with speedups ranging from 1.03–1.17× at 128 bytes and generally increase with row size, reaching 2.99× for sequential scan, 2.34× for hash join, 1.88× for merge join, 1.78× for nested loop join, and 1.52× for indexed nested loop join at 6144 bytes. Sequential scan benefits most because array translation preserves locality across consecutive page identifiers, whereas dependent B-tree probes limit the gain for indexed nested loop join. This trend is consistent with the analysis in Section 3.

6.5 Translation Memory Overhead

We evaluate translation memory consumption using TPC-C (100 warehouses, 18GB–800GB), YCSB-C (614GB), and YCSB-D (800GB, read-latest workload). We compare **Calico** with hole punching, **predicache**, which uses a lock-free hash table but stores 40-byte translation entries, and **vmcache**. Buffer pool configurations are 16GB and 128GB for TPC-C, and 16GB for YCSB-C and YCSB-D. predicache's translation memory scales with buffer pool size rather than database size. For a buffer pool with N frames, predicache provisions its table at $N \times 40 \times 2$ bytes, where 40 bytes is the entry size and the factor of 2 maintains a 50% load factor. We account for *all* memory used for translation state. For predicache this includes the hash table itself. For vmcache this includes OS page tables in addition to the state array used to track resident pages. For Calico, we include the translation array and the metadata for hole punching.

Figure 11 shows translation memory as database size grows. predicache remains constant at 320/2560MB for 16GB/128GB pools. vmcache grows linearly with database size. Calico's hole-punching mechanism reclaims memory for cold regions by exploiting temporal locality. For TPC-C with a 16GB pool, Calico reclaims up to 86.2% of translation memory, which is below predicache's 320 MB despite covering the full logical page space. At the 128GB pool,



(a) TPC-C, 16GB Pool **(b)** TPC-C, 128GB Pool

(c) YCSB-C(614GB), 16GB Pool **(d)** YCSB-D, 16GB Pool

Figure 11: Translation Memory Overhead – Calico's hole punching keeps translation space proportional to working set in TPC-C and YCSB-D. YCSB-C shows challenging behavior with spatially dispersed hot keys that prevent hole punching, yet Calico still uses 0.50× the memory of vmcache.

reclamation drops to 60.4%, but Calico still uses only 632.80 MB, far below predicache's 2,560 MB. YCSB-D shows the strongest hole punching. As new records become the hottest pages, older insertions turn cold and Calico reclaims 95.7% of the translation memory. This is 0.21× predicache and 46.9× smaller than vmcache. In contrast, YCSB-C is the worst case. Its read-only workload and Zipfian distribution spread hot keys throughout the key space, preventing hole punching from forming contiguous cold regions. Nevertheless, Calico is only half of vmcache's memory footprint.

6.6 Ablation Study

To isolate the impact of array translation and system-level optimizations, we perform an ablation study using in-memory DEEP10M HNSW workloads with 12GB buffer pool. We incrementally enable each optimization on top of the baseline Linear Probing Hash translation. The results are shown in Figure 12. Replacing Linear Probing Hash (3.51 kQPS) with Array translation (5.47 kQPS) provides the 1.56× gain. Adding the upper-level mapping required for sparse PIDs reduces throughput to 4.99 kQPS. Path Cache then improves it to 5.36 kQPS, recovering 77% of this loss and leaving a 1.98% gap to Array translation. The remaining gains come from system-level optimizations rather than translation structure alone. Huge-page-backed frames provide the largest step, reaching 7.60 kQPS; huge pages for translation state, optimistic reads, and group prefetch then raise throughput to 7.84, 8.64, and 9.40 kQPS, respectively. The final configuration reaches 2.68× the Linear Probing Hash throughput.

Why Group Prefetch Helps Array Translation More. To isolate the impact of group prefetch, we run an in-memory graph BFS benchmark on a 5M-node graph and compare no prefetch against group prefetch for Calico's array translation, Linear Probing Hash translation, and predicache. Table 5 shows that group prefetch materially helps only array translation, while hash-based designs and vmcache see little or negative gain. For Calico, prefetch lowers

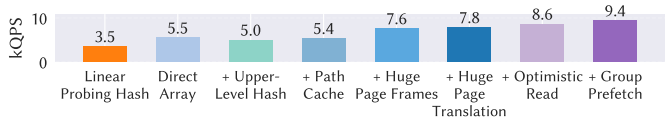
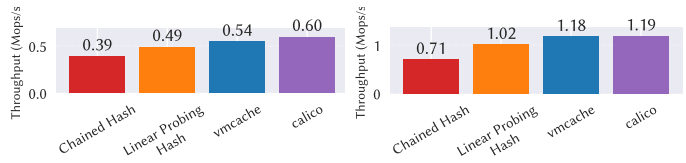


Figure 12: Ablation Study (HNSW DEEP10M, 0.88 Recall@10) – Cumulative throughput of the incremental translation and memory-management optimizations. Array translation provides the main gain over the Linear Probing Hash baseline; the Path Cache optimization compensated the overhead of hierarchical mapping. Huge pages, optimistic reads, and group prefetch provide the remaining improvements, totaling 2.68× the hash-baseline throughput.

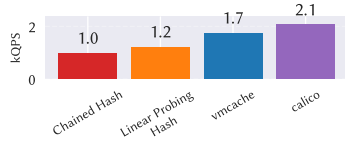
Table 5: Microarchitecture Analysis of Group Prefetch – In-memory graph BFS traversal time on a 5M-node graph, comparing no prefetch vs. group prefetch for various buffer pool designs. Speedup is computed from average traversal time.

Mechanism	Speedup	Cycles/op	L3-stall/op	LLC-miss/op
Calico	1.201×	4903→4082	5521→4271	84.3→65.8
Linear Probing Hash	1.008×	11774→11680	13736→12835	158.7→205.8
predicache	0.946×	11201→11840	11310→11870	309.9→173.9
vmcache(4KB)	0.916×	5410→5905	6295→6650	93.7→98.0
vmcache(2MB)	0.940×	3657→3878	3958→4037	63.0→71.9

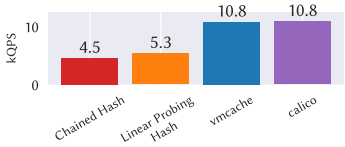


(a) ARM Cortex-A72 (B-tree)

(b) Intel Core i9-13900K (B-tree)



(c) ARM Cortex-A72 (HNSW)



(d) Intel Core i9-13900K (HNSW)

Figure 13: Cross-Platform Validation using Single-threaded B-tree and HNSW vector search – Calico consistently outperforms hash tables and matches or exceeds vmcache across ARM and Intel platforms.

cycles, L3 stall time, and LLC misses enough to produce a clear end-to-end speedup. In Calico, each PID resolves to one array entry, so high-fan-out graph traversal exposes many independent translation loads that software prefetch can overlap. In hash-table and predicache designs, each prefetch still incurs hash table probing and possible collision handling before reaching translation state. These extra instructions and control dependencies reduce effective memory-level parallelism, so the overhead of prefetch is not repaid by comparable latency hiding. vmcache likewise benefits little because translation state lookup is not the dominant dependent bottleneck on this workload.

6.7 Cross-Platform Validation

To validate that Calico's benefits generalize across CPU architectures, we conducted single-threaded experiments on ARM Cortex-A72 and Intel Core i9-13900K using HNSW vector search with SIFT1M dataset and YCSB-C workloads with 100M 128-byte records.

HNSW Vector Search (Figures 13c and 13d): On ARM, Calico achieves 2 kQPS, outperforming vmcache by 18% and chained hash by 2.1×. On Intel, Calico reaches 10.8 kQPS, matching vmcache and exceeding chained hash by 2.4×. The larger ARM gap (18% vs. 1%) suggests ARM is more sensitive to TLB overhead.

B-tree YCSB-C On ARM (Figures 13a and 13b), Calico achieves 0.6 Mops/s, outperforming vmcache (4KB pages) by 10% and Linear Probing Hash by 22%. On Intel, Calico reaches 1.19 Mops/s, exceeding Linear Probing Hash by 17%. Calico's huge pages (2MB) eliminate TLB pressure that vmcache faces with 4KB pages, while direct array indexing consistently outperforms hash tables (17–22%) across both shallow (ARM) and deep (Intel) pipelines.

7 RELATED WORK

Main-Memory DBMS and Buffer Management. Main-memory database systems [15, 35, 36, 77] manage data at tuple granularity and have been extended to larger-than-memory datasets [14, 18, 76]. Anti-Caching [14] keeps indexes in memory while evicting cold tuples, limiting scalability [85]. Recent buffer management research explored persistent memory [78, 88], remote memory [24, 71, 92], translation with SIMD hash table [46], or improving memory utilization under skew [89, 91]. Calico complements these approaches: its direct array translation delivers near-hardware translation performance at page granularity, enabling fine-grained eviction control without kernel modifications or eschewing buffer manager. Hashing research has also improved cache behavior and skewed-load handling, e.g., VIP hashing [34]; these optimizations improve hash-table lookups but retain stored keys, probing, and PID randomization on the buffer-manager translation path.

Indirection and Virtual Memory. Prior systems use indirection arrays, mapping tables, or virtual-memory mechanisms at different layers and for different objectives. Bw-tree [45, 82] and Bf-tree [23] maintain index-local mapping tables from logical tree node IDs to memory pointers or disk offsets to support latch-free updates, delta chains, or mini-page buffering. Those live mappings remain non-zero, so their arrays scale with index size and cannot exploit Calico's all-zero evicted state for lazy allocation, shared-zero-page backing, or hole punching. ERMIA [37] uses indirection arrays for record-level MVCC by mapping object IDs to version chains. RUMA [72] rewires OS page mappings to build flexible memory regions for resizing, partitioning, and snapshotting. Calico instead provides a DBMS-wide user-space buffer-pool translation layer that is agnostic to data structure design, enabling broad applicability across diverse workloads.

Software Prefetching. Software prefetching has been widely studied to hide memory and IO latency in various contexts, including database query processing [11, 32, 69] and transaction processing [25, 28]. Unlike existing work, Calico's group prefetch interface allows modern workloads such as graph-based vector search to express memory-level/IO parallelism to the buffer manager.

Larger-Than-Memory Vector Search. Parallel SSD I/O has been exploited to improve disk-based vector search [10, 30, 73, 81]. Unlike existing work, Calico focuses on improving in-memory performance of buffer-managed vector search index and I/O performance under memory pressure through group prefetch.

OS-Level Memory Management. Recent research has explored improving memory management through OS-level mechanisms. Enhanced OS paging systems [7, 21, 29, 53, 64, 80, 87] aim to improve performance for larger-than-memory workloads, but operate at hardware page table granularity and face the eviction granularity problem. DBMS/OS co-design approaches such as libdbos [90] and CumulusDB [41] explore running database systems in privileged kernel space to enable more powerful abstractions for buffer management and snapshotting. In contrast, Calico demonstrates that careful user-space design can achieve hardware-page-table performance with fine-grained eviction without kernel modifications.

8 CONCLUSION

This paper presents Calico, a novel DBMS-managed buffer pool that revives array-based translation, combining huge-page-backed frames, multi-level translation, hole punching, and group prefetch into a single design that is strong on OLTP, OLAP, and vector search workloads, and larger-than-memory I/O while remaining deployable in user space. Our evaluation shows that Calico delivers 2–6× higher throughput than page-table-based designs under memory pressure, 1.24× speedup over the strongest hash baseline on B-tree workloads, and up to 3.95× in-memory and 6.5× out-of-memory end-to-end speedup in PostgreSQL/pgvector, achieving hardware-competitive resident-page performance while retaining full DBMS control over eviction and I/O.

References

[1] 2024. Datasets for approximate nearest neighbor search. http://corpus-texmex.irisa.fr.

[2] 2024. madvise Linux manual page. https://man7.org/linux/man-pages/man2/madvise.2.html.

[3] 2024. RocksDB's BlockCache key format. https://github.com/facebook/rocksdb/blob/main/cache/cache_key.cc.

[4] 2024. sys.dm_os_buffer_descriptors (Transact-SQL). https://learn.microsoft.com/en-us/sql/relational-databases/system-dynamic-management-views/sys-dm-os-buffer-descriptors-transact-sql?view=sql-server-ver17.

[5] 2024. WiredTiger's source tree. https://github.com/wiredtiger/wiredtiger.

[6] 2025. Linux kernel implementation of MADV_DONTNEED. https://elixir.bootlin.com/linux/v6.17.7/source/mm/madvise.c.

[7] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can far memory improve job throughput?. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[8] William Bridge, Ashok Joshi, M Keihl, Tirthankar Lahiri, Juan Loaiza, and N MacNaughton. 1997. The oracle universal server buffer manager. In *PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES*. INSTITUTE OF ELECTRICAL & ELECTRONICS ENGINEERS (IEEE), 590–594.

[9] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, 209–223.

[10] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. SPANN: Highly-efficient Billion-scale Approximate Nearest Neighborhood Search. *Advances in Neural Information Processing Systems* 34 (2021), 5199–5212.

[11] Shimin Chen, Anastassia Ailamaki, Phillip B Gibbons, and Todd C Mowry. 2007. Improving hash join performance through prefetching. *ACM Transactions on Database Systems (TODS)* 32, 3 (2007), 17–es.

[12] Andrew Crotty, Viktor Leis, and Andrew Pavlo. 2022. Are You Sure You Want to Use MMAP in Your Database Management System?. In *Conference on Innovative Data Systems Research (CIDR 2022)*. https://www.cidrdb.org/cidr2022/papers/p13-crotty.pdf

[13] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2013. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 33–48.

[14] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik. 2013. Anti-caching: A New Approach to Database Management System Architecture. *Proc. VLDB Endow.* 6, 14 (Sept. 2013), 1942–1953.

[15] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *SIGMOD*. 1243–1254.

[16] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The Faiss library. arXiv preprint arXiv:2401.08281. https://arxiv.org/abs/2401.08281

[17] Wolfgang Effelsberg and Theo Haerder. 1984. Principles of database buffer management. *ACM Transactions on Database Systems (TODS)* 9, 4 (1984), 560–595.

[18] Ahmed Eldawy, Justin Levandoski, and Per-Åke Larson. 2014. Trekking through siberia: Managing cold data in a memory-optimized database. *PVLDB* 7, 11 (2014), 931–942.

[19] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2017. Fast approximate nearest neighbor search with the navigating spreading-out graph. *arXiv preprint arXiv:1707.00143* (2017).

[20] Goetz Graefe, Haris Volos, Hideaki Kimura, Harumi Kuno, Joseph Tucek, Mark Lillibridge, and Alistair Veitch. 2014. In-Memory Performance for Big Data. *VLDB* 8, 1 (2014).

[21] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. 2017. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 649–667.

[22] Annamalai Gurusami. 2015. Extent Descriptor Page of InnoDB. https://dev.mysql.com/blog-archive/extent-descriptor-page-of-innodb/.

[23] Xiangpeng Hao and Badrish Chandramouli. 2024. Bf-tree: A modern read-write-optimized concurrent larger-than-memory range index. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3442–3455.

[24] Xiangpeng Hao, Xinjing Zhou, Xiangyao Yu, and Michael Stonebraker. 2024. Towards buffer management with tiered main memory. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–26.

[25] Yongjun He, Jiacheng Lu, and Tianzheng Wang. 2020. CoroBase: coroutine-oriented main-memory database engine. *arXiv preprint arXiv:2010.15981* (2020).

[26] Gavin Henry. 2019. Howard chu on lightning memory-mapped database. *Ieee Software* 36, 06 (2019), 83–87.

[27] Hironobu SUZUKI. 2024. PostgreSQL Buffer Tag. https://www.interdb.jp/pg/pgsql08/01.html.

[28] Kaisong Huang, Tianzheng Wang, Qingqing Zhou, and Qingzhong Meng. 2023. The art of latency hiding in modern database engines. *Proceedings of the VLDB Endowment* 17, 3 (2023), 577–590.

[29] Sepehr Jalalian, Shaurya Patel, Milad Rezaei Hajidehi, Margo Seltzer, and Alexandra Fedorova. 2024. ExtMem: Enabling Application-Aware Virtual Memory Management for Data-Intensive Applications. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, 397–408.

[30] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in neural information processing Systems* 32 (2019).

[31] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence* 33, 1 (2010), 117–128.

[32] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. 2018. Exploiting coroutines to attack the "killer nanoseconds". *Proceedings of the VLDB Endowment* 11, 11 (2018), 1702–1714. doi:10.14778/3236187.3236216

[33] Jonathan Corbet. 2018. The final step for huge-page swapping. https://lwn.net/Articles/758677/.

[34] Aarati Kakaraparthy, Jignesh M Patel, Brian P Kroth, and Kwanghyun Park. 2022. VIP hashing: adapting to skew in popularity of data on the fly. *Proceedings of the VLDB Endowment* 15, 10 (2022).

[35] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. In *VLDB*. 1496–1499.

[36] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE (ICDE)*. 195–206.

[37] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. Ermia: Fast memory-optimized database system for heterogeneous workloads.

In *Proceedings of the 2016 International Conference on Management of Data*. 1675–1687.
[38] Leonardo Kuffo, Elena Krippner, and Peter Boncz. 2025. PDX: A data layout for vector similarity search. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–26.
[39] Viktor Leis. 2024. LeanStore: A High-Performance Storage Engine for NVMe SSDs. *Proceedings of the VLDB Endowment* 17, 12 (2024), 4536–4545.
[40] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. 2023. Virtual-Memory Assisted Buffer Management. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–25.
[41] Viktor Leis and Christian Dietrich. 2024. Cloud-Native Database Systems and Unikernels: Reimagining OS Abstractions for Modern Hardware. *PVLDB* 17 (2024).
[42] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-memory data management beyond main memory. In *ICDE*. IEEE, 185–196.
[43] Daniel Lemire. 2018. Measuring the memory-level parallelism of a system using a small C++ program? https://lemire.me/blog/2018/11/05/measuring-the-memory-level-parallelism-of-a-system-using-a-small-c-program/.
[44] Daniel Lemire. 2025. Memory-level parallelism: Apple M2 vs Apple M4. https://lemire.me/blog/2025/07/09/memory-level-parallelism-apple-m2-vs-apple-m4/.
[45] Justin J Levandoski, David B Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *ICDE*. 302–313.
[46] Mingyu Liu, Junbin Kang, Kai Wang, Lu Zhang, Haibo Chen, Xiuchang Li, and Tianhong Ding. 2025. ScaleCache: Scalable and Production-Grade Buffer Management for Disk-Based Database Systems. *Proceedings of the VLDB Endowment* 18, 12 (2025), 5073–5085.
[47] Duo Lu, Helena Caminal, Manos Chatzakis, Yannis Papakonstantinou, Yannis Chronis, Vaibhav Jain, and Fatma Özcan. 2026. An In-Depth Study of Filter-Agnostic Vector Search on a PostgreSQL Database System: [Experiments & Analysis]. *Proceedings of the ACM on Management of Data* 4, 3 (SIGMOD), Article 134 (2026), 26 pages. doi:10.1145/3802011
[48] LWN.net. 2025. Introducing support for huge zero pages. https://lwn.net/Articles/1033058/.
[49] Fabian Mahling, Marcel Weisgut, and Tilmann Rabl. 2025. Fetch Me If You Can: Evaluating CPU Cache Prefetching and Its Reliability on High Latency Memory. In *Proceedings of the 21st International Workshop on Data Management on New Hardware*. 1–9.
[50] Yu A Malkov and Dmitry A Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42, 4 (2020), 824–836. doi:10.1109/TPAMI.2018.2889473
[51] Aninda Manocha, Zi Yan, Esin Tureci, Juan L Aragón, David Nellans, and Margaret Martonosi. 2023. Architectural support for optimizing huge page selection within the OS. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 1213–1226.
[52] Paul E McKenney. 2010. Memory barriers: a hardware view for software hackers. *Linux Technology Center, IBM Beaverton* (2010).
[53] Ilya Meignan-Masson, Masanori Misono, Viktor Leis, and Pramod Bhatotia. 2026. uCache: A Customizable Unikernel-based IO Cache. In *24th USENIX Conference on File and Storage Technologies (FAST 26)*. USENIX Association, 701–720.
[54] Theodore Michailidis, Alex Delis, and Mema Roussopoulos. 2019. Mega: Overcoming traditional problems with os huge page management. In *Proceedings of the 12th ACM International Conference on Systems and Storage*. 121–131.
[55] Microsoft. 2025. SQLServer Index architecture and design guide. https://learn.microsoft.com/en-us/sql/relational-databases/sql-server-index-design-guide?view=sql-server-ver17#index-placement-on-filegroups-or-partitions-schemes.
[56] Microsoft. 2025. VirtualAlloc function (memoryapi.h). https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualalloc.
[57] Vivek Narasayya, Ishai Menache, Mohit Singh, Feng Li, Manoj Syamala, and Surajit Chaudhuri. 2015. Sharing buffer pool memory in multi-tenant relational database-as-a-service. *Proceedings of the VLDB Endowment* 8, 7 (2015), 726–737.
[58] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf
[59] Oracle. 2025. MySQL. http://mysql.com.
[60] Oracle. 2025. MySQL btree implementation. https://github.com/mysql/mysql-server/blob/trunk/storage/innobase/btr/btr0cur.cc#L883.
[61] Oracle. 2025. Oracle Index Storage. https://docs.oracle.com/en/database/oracle/oracle-database/26/cncpt/indexes-and-index-organized-tables.html#GUID-832C2B66-912B-4E1C-B4B5-AA40F49E4970.
[62] Oracle. 2026. Bigfile Tablespaces in Oracle Database. https://docs.oracle.com/en/database/oracle/oracle-database/26/admin/managing-tablespaces.html#GUID-7E376766-D99D-4274-BF00-8DC1E6FC5063.
[63] Riki Otaki, Jun Hyuk Chang, Aaron J. Elmore, and Goetz Graefe. 2025. Enhancing Transaction Processing through Indirection Skipping. *Proc. VLDB Endow.* 18, 11 (Sept. 2025), 4104–4116. doi:10.14778/3749646.3749680
[64] Anastasios Papagiannis, Manolis Marazakis, and Angelos Bilas. 2021. Memory-mapped I/O on steroids. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 277–293.
[65] pgvector contributors. 2025. pgvector hnsw graph traversal. https://github.com/pgvector/pgvector/blob/master/src/hnswutils.c#L818.
[66] PostgreSQL Global Development Group. 2025. PostgreSQL. https://www.postgresql.org/.
[67] PostgreSQL Global Development Group. 2025. PostgreSQL btree implementation. https://github.com/postgres/postgres/blob/master/src/backend/access/nbtree/nbtsearch.c#L107.
[68] PostgreSQL Global Development Group. 2025. PostgreSQL Database File Layout. https://www.postgresql.org/docs/current/storage-file-layout.html.
[69] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. 2017. Interleaving with coroutines: a practical approach for robust index joins. *Proceedings of the VLDB Endowment* 11, 2 (2017), 230–242.
[70] RavenDB. 2025. RavenDB's source tree. https://docs.ravendb.net/4.0/server/storage/storage-engine/.
[71] Niklas Riekenbrauck, Marcel Weisgut, Daniel Lindner, and Tilmann Rabl. 2024. A three-tier buffer manager integrating CXL device memory for database systems. In *2024 IEEE 40th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 395–401.
[72] Felix Martin Schuhknecht, Jens Dittrich, and Ankur Sharma. 2016. RUMA Has It: Rewired User-space Memory Access is Possible! *Proceedings of the VLDB Endowment* 9, 10 (2016), 768–779.
[73] Joobo Shim, Jaewon Oh, Hongchan Roh, Jaeyoung Do, and Sang-Won Lee. 2025. Turbocharging Vector Databases using Modern SSDs. *Proceedings of the VLDB Endowment* 18, 11 (2025), 4710–4722.
[74] SQLite Development Team. 2025. sqlite. https://github.com/sqlite/sqlite/blob/master/src/pcache1.c.
[75] sqlite-vec contributors. 2025. A vector search SQLite extension that runs anywhere! https://github.com/asg017/sqlite-vec.
[76] Radu Stoica and Anastasia Ailamaki. 2013. Enabling Efficient OS Paging for Main-Memory OLTP Databases. In *DaMon*. 7 pages.
[77] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*. 1150–1160.
[78] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing Non-Volatile Memory in Database Systems. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. Association for Computing Machinery, 1541–1555. doi:10.1145/3183713.3196897
[79] Ash Vardanian. 2023. *USearch by Unum Cloud*. doi:10.5281/zenodo.7949416
[80] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2020. Semeru: A Memory-Disaggregated Managed Runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 261–280.
[81] Mengzhao Wang, Weizhi Xu, Xiaomeng Yi, Songlin Wu, Zhangyang Peng, Xiangyu Ke, Yunjun Gao, Xiaoliang Xu, Rentong Guo, and Charles Xie. 2024. Starling: An i/o-efficient disk-resident graph index framework for high-dimensional vector similarity search on data segment. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–27.
[82] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G Andersen. 2018. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*. 473–488.
[83] Seth John White. 1994. *Pointer swizzling techniques for object-oriented database systems*. The University of Wisconsin-Madison.
[84] Yandex Research. 2024. Benchmarks for Billion-Scale Similarity Search. https://research.yandex.com/blog/benchmarks-for-billion-scale-similarity-search.
[85] Huanchen Zhang, David G Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the storage overhead of main-memory OLTP databases with hybrid indexes. In *SIGMOD*. 1567–1581.
[86] Yunan Zhang, Shige Liu, and Jianguo Wang. 2024. Are there fundamental limitations in supporting vector data management in relational databases? A case study of PostgreSQL. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 3640–3653.
[87] Kan Zhong, Wenlin Cui, Xin Chen, Qiao Li, Zhe Yang, Youyou Lu, Xiaodan Yan, Siwei Luo, Qizhao Yuan, and Keji Huang. 2023. Revisiting swapping in user-space with lightweight threading. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 11 (2023), 4205–4218.
[88] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David Cohen. 2021. Spitfire: A Three-Tier Buffer Manager for Volatile and Non-Volatile Memory. In *Proceedings of the 2021 International Conference on Management of Data*. 2195–2207.
[89] Xinjing Zhou, Xiangpeng Hao, Xiangyao Yu, and Michael Stonebraker. 2025. Tiered-Indexing: Optimizing Access Methods for Skew. *The VLDB Journal* 34, 4 (2025), 45. doi:10.1007/s00778-025-00928-6

[90] Xinjing Zhou, Viktor Leis, Jinming Hu, Xiangyao Yu, and Michael Stonebraker. 2025. Practical db-os co-design with privileged kernel bypass. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–27.

[91] Xinjing Zhou, Xiangyao Yu, Goetz Graefe, and Michael Stonebraker. 2023. Two is Better Than One: The Case for 2-TREE for Skewed Data Sets. In *13th Annual Conference on Innovative Data Systems Research (CIDR 2023)*. https://www.cidrdb.org/cidr2023/papers/p57-zhou.pdf

[92] Tobias Ziegler, Carsten Binnig, and Viktor Leis. 2022. ScaleStore: A fast and cost-efficient storage engine using DRAM, NVMe, and RDMA. In *Proceedings of the 2022 International Conference on Management of Data*. 685–699.

[93] Michael Zinsmeister, Lam-Duy Nguyen, Viktor Leis, and Thomas Neumann. 2026. Predictive Translation: High-Performance Buffer Management Without the Trade-Offs. *Proc. ACM Manag. Data* 4, 1, Article 64 (April 2026), 16 pages. doi:10.1145/3786678