TIDEHUNTER: Large-Value Storage With Minimal Data Relocation

Andrey Chursin
Mysten Labs
andrey@mystenlabs.com

Lefteris Kokoris-Kogias
Mysten Labs
lefteris@mystenlabs.com

Alex Orlov
Mysten Labs
alexo@mystenlabs.com

Alberto Sonnino
Mysten Labs and UCL
alberto@mystenlabs.com

Igor Zablotchi
Mysten Labs
igor@mystenlabs.com

ABSTRACT

Log-Structured Merge-Trees (LSM-trees) dominate persistent key-value storage but suffer from high write amplification from 10× to 30× under random workloads due to repeated compaction. This overhead becomes prohibitive for large values with uniformly distributed keys, a workload common in content-addressable storage, deduplication systems, and blockchain validators. We present TIDEHUNTER, a storage engine that eliminates value compaction by treating the Write-Ahead Log (WAL) as permanent storage rather than a temporary recovery buffer. Values are never overwritten; and small, lazily-flushed index tables map keys to WAL positions. TIDEHUNTER introduces (a) lock-free writes that saturate NVMe drives through atomic allocation and parallel copying, (b) an optimistic index structure that exploits uniform key distributions for single-roundtrip lookups, and (c) background relocation that reclaims space without blocking writes. On a 1 TB dataset with 1 KB values, TIDEHUNTER achieves 763K writes per second, that is 6.8× higher than RocksDB and 2.2× higher than BlobDB, while improving point queries by 2× and existence checks by 21.1×. We validate real-world impact by integrating TIDEHUNTER into Sui, a high-throughput blockchain, where it maintains stable throughput and latency under loads that cause RocksDB-backed validators to collapse. TIDEHUNTER is production-ready and is being deployed in production within Sui.

PVLDB Reference Format:
Andrey Chursin, Lefteris Kokoris-Kogias, Alex Orlov, Alberto Sonnino, and Igor Zablotchi. TIDEHUNTER: Large-Value Storage With Minimal Data Relocation. PVLDB, 19(11): 3786 - 3798, 2026.
doi:10.14778/3836663.3836725

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/MystenLabs/tidehunter/tree/paper-experiments.

1 INTRODUCTION

Log-Structured Merge-Trees (LSM-trees) dominate persistent key-value store design. An LSM-tree buffers writes in memory, flushes them to disk as sorted files, and periodically merges (compacts) those files to bound their count and keep lookups fast. For small values, the rewriting overhead is acceptable; for values of several kilobytes, it is not. Write amplification (bytes written to disk divided by bytes from the application) reaches 10× to 30× [20] under random workloads and grows with dataset size. A store ingesting 100 MB/s of application data may push 1 GB/s to disk, starving reads of bandwidth.

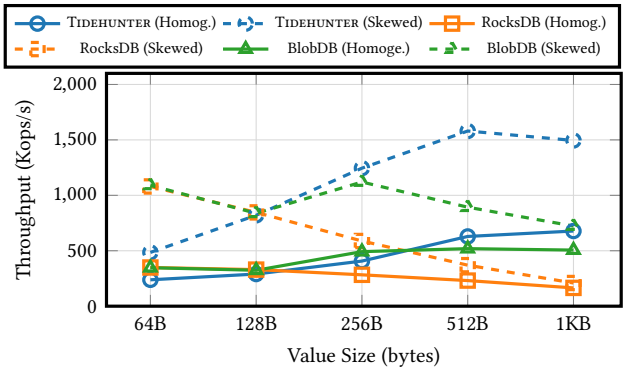


Figure 1: Mixed workload throughput (50% reads/writes) vs. value size, with skewed (Zipf $\theta = 2$) and homogeneous (Zipf $\theta = 0$) access patterns.

LSM-trees were designed for hard disk drives, where sequential I/O is 100× faster than random and trading write amplification for sequential access made sense. Solid-state drives (SSDs) changed the equation: sequential-to-random gaps shrink to 10× or less for large requests, and SSDs expose internal parallelism that LSM-trees cannot exploit. SSD cells also wear out after a finite number of writes, so high write amplification accelerates device failure.

WiscKey [29] introduced key-value separation: keys stay in an LSM-tree, values go to a separate append-only log. Values stop moving during compaction, and write amplification drops. However, the LSM-tree remains, and so do its compaction cost and multi-level lookup overhead. Under uniform key distributions, lookups traverse multiple levels with little cache benefit, and WiscKey's value log needs its own garbage collection, which stalls I/O.

Some designs pair an append-only log with a hash index to avoid LSM compaction. Bitcask [44] and FAWN-DS [1] use in-memory hash indexes for fast lookups, but their index must fit in RAM, thus bounding dataset size and slowing down startups. CedrusDB [47] uses a memory-mapped lazy-trie for on-disk access and faster recovery, but this still requires multiple disk accesses to locate keys.

Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836725

We introduce Tidehunter, in which the Write-Ahead Log (WAL), traditionally a temporary crash-recovery buffer, becomes the permanent store for values. We replace the key LSM-tree with sharded index tables that flush lazily. Unlike in-memory hash indexes, Tidehunter caches indices on demand, sidestepping memory limits and avoiding full loads at startup. Moreover, rather than traversing an on-disk data structure, it directly estimates a key's on-disk index position.

Our design targets applications with large values and uniformly distributed keys. Content-addressable storage systems like Git and IPFS key objects by cryptographic hashes of their contents [5], and object stores like Facebook's Haystack key billions of 8–64 KB photos by UUID [4]. Similar patterns appear in deduplication systems [22, 49] and machine-learning feature stores [15, 17, 39]. In all these settings, keys lack locality by design, and values are large enough that LSM-tree compaction becomes the dominant cost.

We focus on blockchain validators as a concrete evaluation target because they combine all of these pressures in a single system: hash-keyed data, kilobyte values, write-heavy ingestion, latency-sensitive reads, and aggressive pruning. We integrate Tidehunter into Sui [7], a high-throughput blockchain whose validators handle thousands of transactions per second and serve concurrent queries over history, execution results, and object state. Validators write continuously as blocks arrive but must read with low latency, and they prune aggressively, keeping only recent epochs (bounded retention windows). This stresses the underlying RocksDB store, where ill-timed compactions can collapse throughput.

Tidehunter is built for large values and uniform keys. Figure 1 shows its throughput advantage growing with value size, under both skewed and homogeneous (uniformly random) access. Three insights guided our design:

(1) **Separate values from indices.** Values live in a central WAL; indices are small and separate. Compaction rewrites indices, not values. Large values are written once and never copied.
(2) **Manage indices lazily.** Indices load on-demand and unload under memory pressure, enabling working sets larger than RAM without manual tuning.
(3) **Exploit uniform key distribution.** When keys are hashes or UUIDs, their distribution is known a priori. Tidehunter estimates a key's on-disk index position directly from the key bits, resolving lookups in a single I/O without trie or tree traversal and without training a model.

We evaluate Tidehunter against RocksDB and BlobDB (the production implementation of WiscKey) on 1 TB datasets. At 1 KB values, Tidehunter achieves 763K writes/sec (up to 6.8× faster than RocksDB and 2.2× faster than BlobDB), improves point queries by 2.0×, and speeds up existence checks by 21.1× (resolved from the index, no value fetch). For small 64-byte values on homogeneous workloads, LSM-trees remain 1.4× to 2.2× faster.

To validate that synthetic benchmarks translate to real systems, we run Sui validators under sustained transaction load. With RocksDB, validators degrade at around 6,000 transactions per second as disk utilization rises and latency climbs. Under the same load, Tidehunter maintains stable throughput and lower latency with reduced disk pressure, avoiding the I/O saturation that triggers RocksDB's performance collapse.

Contributions. This paper makes the following contributions:

- We present Tidehunter, a storage engine architecture that eliminates value compaction by treating the WAL as permanent storage, achieving near 1× write amplification for large values.
- We introduce an optimistic index that exploits uniform key distributions to achieve single-roundtrip lookups, trading the generality of LSM-trees for lower tail latencies.
- We evaluate Tidehunter on synthetic benchmarks and a production blockchain, demonstrating up to 6.8× higher write throughput compared to RocksDB for 1 KB values and stable performance where RocksDB-backed systems collapse.

2 BACKGROUND & DESIGN PRINCIPLES

We review LSM-trees and their compaction costs, then derive the principles guiding Tidehunter's design.

2.1 LSM-Trees and Their Costs

LSM-trees [37] buffer writes in an in-memory *memtable*, flush it as an immutable Sorted String Table (SST) to Level 0, and periodically *compact* files downward into larger levels [20]. Each level is typically 10× the size of the previous, giving $O(\log N)$ levels for a dataset of size N; compaction maintains the invariant that files within a level have non-overlapping key ranges, so reads at L1 and beyond touch one file per level.

Compaction maintains read performance but imposes three costs that dominate under write-heavy workloads. *Write amplification*: a record is rewritten at every level it traverses, reaching 10× to 30× under random workloads [20]; Sui validators ingesting 40 MB/s of application data observed 500 MB/s of disk writes (an 12.5× amplification factor). *Read amplification*: positive lookups on cold data incur roughly one disk read per level (five or six for a 1 TB dataset). *Write stalls*: when compaction falls behind, the engine throttles or blocks foreground writes to bound L0, causing latency spikes and unpredictable throughput [3]. These costs compound with dataset and value size; in Sui's testnet, disk I/O utilization reached 80% with the storage engine rewriting unchanged data.

2.2 Tidehunter Design Principles

These limitations, particularly acute for applications with large values and uniform keys, motivate Tidehunter's design. Three principles guide every decision: (1) writes append directly to their long-term position on disk, (2) values are rarely moved, and (3) compaction never blocks writes. We arrive at Tidehunter by starting from a minimal storage system and incrementally addressing its limitations (Figure 2).

Starting from a bare append-only log ($O(1)$ writes but $O(n)$ reads), we add an in-memory index mapping keys to log positions for fast point lookups. Persisting this index to disk allows datasets to exceed RAM. To reclaim space from deletes and overwrites, a background relocator scans the oldest WAL prefix, re-appends live entries to the tail, and drops empty files. Unlike LSM compaction, this relocation runs concurrently with foreground writes without blocking them. Periodic snapshots record the index state and a WAL position. Recovery then loads the snapshot and replays only the WAL suffix instead of the entire log. Finally, we shard the key space into independent cells (contiguous key-range partitions). Each cell

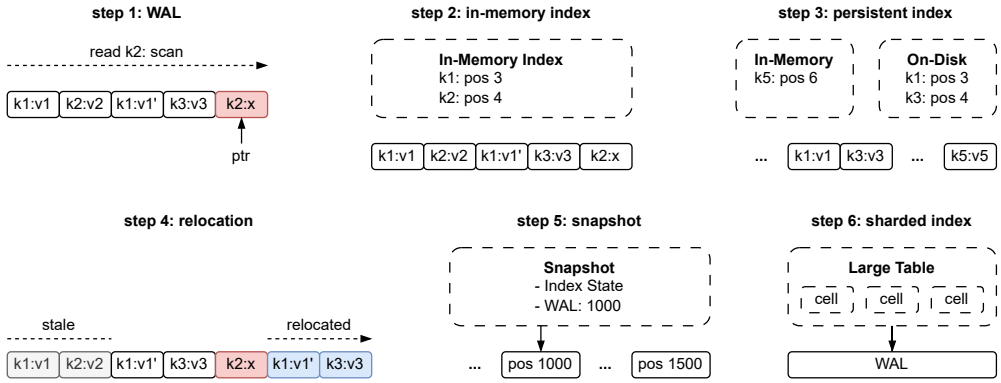


Figure 2: Design evolution of TIDEHUNTER. Each step addresses a limitation of the previous design: (1) append-only log provides fast writes but slow reads; (2) in-memory index enables fast lookups; (3) persisted index supports data larger than RAM; (4) relocation reclaims space without blocking writes; (5) snapshots enable fast recovery; (6) sharding improves concurrency.

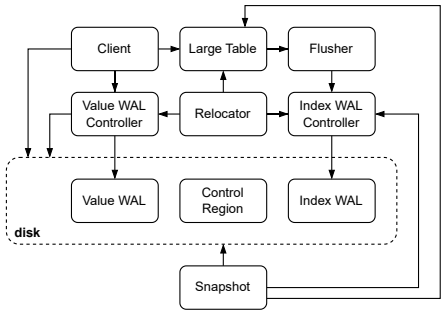


Figure 3: High-level architecture of TIDEHUNTER.

has its own lock and indices to parallelize operations. The WAL remains shared, so writes serialize only during allocation. §3 details each component.

3 THE TIDEHUNTER ARCHITECTURE

Figure 3 illustrates TIDEHUNTER's architecture. This section describes how its components interact to serve reads, writes, and how it does crash recovery.

The system comprises several main components:

- **Large Table**: An in-memory index mapping keys to positions in the *Value WAL*, an append-only log that permanently stores key-value entries.
- **Value WAL Controller**: Manages WAL writes, handling position allocation, memory-mapped I/O, and background durability.
- **Index Store**: An on-disk log storing serialized indices flushed by its controller under memory pressure.
- **Snapshot Engine**: Persists a point-in-time image of the Large Table to the *Control Region* for crash recovery.
- **Relocator**: Reclaims Value WAL space in the background by migrating live entries and discarding obsolete ones.

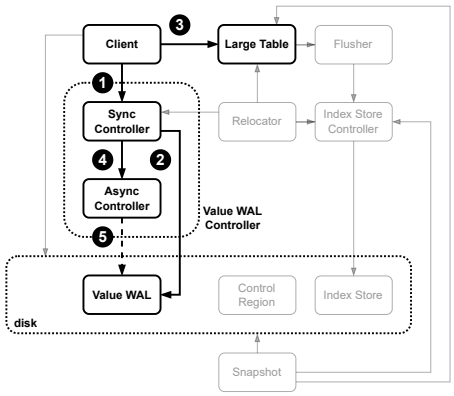


Figure 4: Write path separating fast synchronous operations (solid) from background durability (dotted).

3.1 Write Flow

Figure 4 expands the Value WAL Controller, separating the fast synchronous path (solid arrows) from asynchronous background operations (dotted). The design minimizes latency while providing clear durability guarantees.

The Client flow. The client sends a write operation (insert or delete) to the synchronous controller, which allocates a position in the Value WAL (a contiguous region for the entry) so concurrent writers do not overwrite each other (❶). Next, the client writes the entry directly to the memory-mapped Value WAL (❷). It then updates the Large Table, the in-memory index mapping keys to WAL positions, so subsequent reads can locate the value without scanning (❸). Because the allocation uses a fast atomic counter, multiple clients run the rest of the write flow in parallel.

When the update completes, the asynchronous controller is notified that the position is fully processed (entry written to the WAL and reflected in the index) (❹). It tracks the highest WAL offset

whose preceding entries have all been processed, so the snapshot engine knows which WAL prefix is represented in the Large Table and crash recovery only replays entries beyond it (see §3.4).

The Value WAL is organized into memory-mapped regions called maps (also called fragments), each covering a contiguous range of positions. The asynchronous controller serves three purposes. First, it manages map lifecycle: it maintains a buffer of pre-allocated maps so writes never block on map creation, and finalizes each map once full. Second, it fsyncs finalized maps in the background (❺), keeping this blocking operation off the write path. Third, when relocation advances the minimum required WAL position, it deletes obsolete WAL files no longer needed for recovery (see §4.4).

Durability Guarantees. This design cleanly separates the synchronous write path (allocation, memory-map write, and index update) from background durability. The failure semantics are:

- **After ❷** (memory-map write): The entry is in the OS page cache, but not yet indexed. A process crash is safe because the kernel eventually flushes the page cache, allowing WAL replay to rebuild the index. However, a system crash (e.g., power loss) might lose the data because it is not fsynced.
- **After ❸** (Large Table update): The write returns to the caller. The entry is now *committed*: visible to reads and durable against process crashes, but not system crashes.
- **After ❺** (background flush): The asynchronous controller has fsynced the map. The write is now fully durable against both process and system crashes.

Providing process-crash durability at commit time and system-crash durability asynchronously matches the default behavior of RocksDB [32]. Applications needing immediate system-crash durability can request an explicit flush.

Deletes. Deleting a key writes a tombstone value. On concurrent operations (e.g., a delete and insert for the same key), the operation with the higher WAL position wins, ensuring consistency regardless of Large Table application order. Tombstones serve three purposes: they survive crash recovery (replaying the tombstone removes the key), they enable conflict resolution with concurrent writes, and they let the relocator skip deleted entries when compacting (§4.4).

Atomic batch writes. A batch is accumulated in memory and committed in two phases: a batch-start marker is written to the Value WAL followed by all entries (all under a single Value WAL allocation), and then each entry is applied to the Large Table in sequence. If a crash interrupts after the WAL write but before the index update, recovery replays the batch; the batch-start marker records the expected entry count, so incomplete or corrupted batches are safely discarded for atomic all-or-nothing visibility.

3.2 Read Flow

Figure 5 expands the Large Table from Figure 3: reads follow a tiered lookup strategy.

A read first consults the in-memory LRU cache of recently accessed values (❶); a hit returns immediately, requiring no index lookup or WAL access. On a miss, the read checks a Bloom filter (❷). If it indicates the key is absent, the lookup returns immediately, unlike RocksDB-style LSMs which traverse multiple SSTable levels regardless. Otherwise, the read consults the Large Table (❸). If the

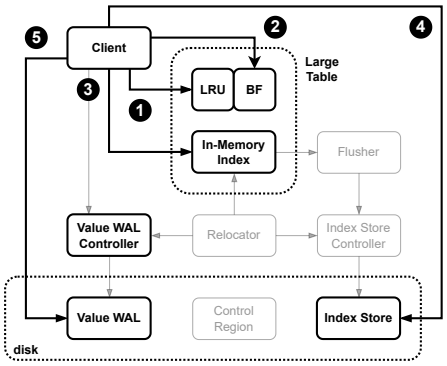


Figure 5: Read path showing the tiered lookup strategy. The snapshot component is omitted for clarity.

relevant index is in memory, the lookup completes immediately; otherwise it loads from the Index Store on disk (❹). The index yields a WAL position, the value is read directly from the Value WAL (❺) and cached in the LRU. In practice, frequently accessed data stays in the OS page cache, making most reads memory-speed.

When a cell's index lives on disk, loading the full index just to fetch one key is wasteful for large cells. Tidehunter instead exploits the workload's uniform key distribution (keys are cryptographic hashes, or prefixed by such hashes) to estimate where a key would appear in the sorted serialized index: treat the key as an integer, compute its fraction of the full key range (a key starting with byte 0x80 sits roughly at the midpoint), and multiply by the index file size to get an estimated byte offset.

The system reads a small window around this offset. If the target falls inside, binary search locates it; otherwise the window shifts based on key comparison and the search repeats. For uniformly distributed keys, this typically converges in 1–3 iterations, reading only a few kilobytes rather than megabytes. We call this the *optimistic index* (it optimistically assumes the key lies within the estimated window, paying extra I/O only when the estimate is off); §4.2 details its statistical foundation. The technique is especially valuable during recovery and cold-start, when most cells are unloaded: Tidehunter can serve reads immediately after opening the database, without waiting for indices to load (see §3.4).

3.3 State Snapshots

Tidehunter periodically snapshots state to bound recovery time. For each cell, a snapshot stores the Index Store position of its most recently flushed index (if any) and a replay position indicating how much of the Value WAL is reflected in that flushed index. The snapshot also records a global replay-from position (the minimum replay position across all cells), which marks where Value WAL replay begins during recovery. A background thread periodically flushes cells that have accumulated significant updates and writes the snapshot metadata atomically to the Control Region, a dedicated on-disk region (Figure 3). More frequent snapshots reduce recovery time but increase write I/O; Tidehunter balances this by flushing only cells whose buffered unpersisted writes exceed a configurable

threshold (cell states are detailed in §4.1). Unlike LSM-trees [32] that snapshot full data structures, Tidehunter only snapshots positions, which are small and cheap to rewrite frequently.

3.4 Crash Recovery

On startup after a crash or shutdown, Tidehunter first reads the Control Region for the most recent snapshot (per-cell Index Store positions and the global replay-from position). It opens the Value WAL and the Index Store; both are append-only logs whose contents survived the crash intact. The Large Table is initialized from the snapshot metadata: each cell starts unloaded but knows where its serialized index resides, so indices load on demand rather than upfront. Starting from the replay-from position, Tidehunter iterates through the Value WAL and replays each entry into the Large Table to reconstruct post-snapshot updates. Finally, it starts the background components (Value WAL Controller, Index Store Controller, flusher threads, relocator, snapshot thread); the database is now ready to serve clients.

4 RESOURCE MANAGEMENT

This section describes how Tidehunter scales beyond available memory: the Large Table organization (partial on-disk residency), index flushing (relieving memory pressure), and relocation (reclaiming disk space without blocking writes).

4.1 The Large Table

The Large Table is Tidehunter's in-memory index, mapping keys to Value WAL positions. Portions of the index may reside on disk, and fine-grained locking supports concurrent access.

Partitioning and concurrency. Two challenges face an index over a dataset larger than memory: the index cannot fit entirely in RAM, and a single lock would serialize all operations. Tidehunter partitions keys into cells, each spanning a contiguous key range, that can be independently loaded, evicted, and locked. Applications define key spaces at database creation, each with a distribution type. Uniformly distributed keys (cryptographic hashes) use a fixed-size cell array; keys with common prefixes use a dynamically growing B-tree, suiting composite keys like (tenant_id, record_id) and enabling efficient prefix scans without pre-allocation. Within each key space, cells are grouped into rows protected by sharded mutexes, allowing concurrent operations on different key ranges to proceed without contention.

Cell states. A naive approach would load an entire cell's index into memory on first access, but this is prohibitive for write-heavy workloads: a single write to a cold cell could trigger loading megabytes of index data. Tidehunter avoids this with a *DirtyUnloaded* state that buffers new writes in memory without loading the existing on-disk index. Each cell maintains one of five states: *Empty* (initial); *Loaded* (fully resident in memory); *Unloaded* (evicted, index only on disk); *DirtyLoaded* (in-memory index with unpersisted changes); and *DirtyUnloaded* (recent writes buffered in memory while the bulk of the index remains on disk). A write to an Unloaded cell transitions it to DirtyUnloaded and buffers only the new entry. Reads check the in-memory buffer first; misses do a point lookup into the on-disk index without loading the full structure (§3.2), trading higher per-lookup cost for lower memory consumption.

4.2 Optimistic Index Lookup

When a cell's index lives on disk, lookups typically require loading the full index or using a multi-level structure with multiple I/Os. Tidehunter avoids both by exploiting uniform key distributions: a key's position in a sorted array can be estimated from its value.

Consider N keys sampled uniformly from $[0, 2^{256})$ and sorted into an array. For a key k, its expected position is $k \cdot N/2^{256}$. More precisely, the i-th order statistic of N uniform samples follows a Beta distribution, $\text{Beta}(i, N - i + 1)$, which provides confidence intervals for the key's location. This statistical foundation allows Tidehunter to bound the search region with high probability.

The on-disk index is a sorted array of fixed-size entries (32-byte key + 12-byte WAL position = 44 bytes), with no headers or directory structures. This enables direct position estimation: for a key k in an index of N entries, the estimated byte offset is $(k/2^{256}) \times 44N$.

To perform a lookup, the system reads a window of W entries (800 by default, about 32 KB) centered at the estimated offset. If the target falls inside, binary search locates it; otherwise the window shifts based on key comparison and the search repeats. For uniformly distributed keys, this typically converges in one to three iterations. Modern SSDs exhibit a batch-read property: reading 32 KB costs essentially the same as reading a byte, due to internal page sizes and command overhead. A window of 800 entries achieves > 99% containment probability for indices of 10K–100K entries. The window size trades hit rate against per-iteration I/O volume; we evaluate this in §6.

4.3 Index Flushing

When a cell accumulates too many dirty entries, it must flush them to disk. Blocking the cell during flush would stall write-heavy workloads, so Tidehunter flushes asynchronously: the flusher snapshots the dirty index and dispatches persistence to background threads while the cell remains available for new writes.

The flusher's action depends on cell state: for DirtyLoaded cells it serializes the complete in-memory index; for DirtyUnloaded cells it first loads the existing on-disk index, merges in the dirty entries, and serializes the combined result. It may also drop obsolete entries. The prepared index is handed to the Index Store Controller for persistence. The Index Store shares the same append-only implementation as the Value WAL (§3.1), differing only in the data stored (serialized indices rather than key-value entries). The controller notifies the Large Table with the new index position, and the cell unmerges: entries included in the flush are removed from the in-memory buffer, leaving only those that arrived after flush start. If none remain, the cell becomes Unloaded; otherwise it stays DirtyUnloaded with the updated on-disk pointer.

Reads during a flush are safe because the Index Store is append-only: new indices are appended rather than overwriting. The cell keeps pointing to the previous index position until the flusher atomically swaps the pointer after the new index is written. Readers and writers thus operate on disjoint regions, requiring no coordination beyond the pointer swap.

4.4 Relocation

As writes accumulate, obsolete entries (overwritten values and deleted keys) consume Value WAL space that simple truncation cannot reclaim. *Relocation* reclaims this space by moving live entries forward in the WAL, allowing old files to be deleted: scan the WAL, identify still-live entries, re-write them at the tail, update the Large Table to point to the new positions, and delete the old files. The challenge is correctness under concurrent writes: a relocated entry must not overwrite a newer update. Tidehunter uses a compare-and-set mechanism. When relocation reads an entry at position P and prepares to relocate it to P', it captures the last processed value L (the highest contiguous WAL position whose preceding writes have all been written and indexed). When applying the update, it only replaces the index entry if the current position is less than L. If a concurrent write updated the key to position $P'' > L$, the relocation update is ignored and the index continues to point to P''. This lets relocation run concurrently with normal writes without locks. Space is reclaimed at file granularity. The Value WAL is organized as a sequence of files; once all live entries in a file have been relocated (or were already beyond the file's range), the file can be deleted. The GC watermark tracks the oldest position that may still be referenced, preventing premature deletion.

Tidehunter supports two relocation strategies. *WAL-based relocation* scans the Value WAL sequentially from the oldest entry; for each, it checks whether the index still references this WAL position, and if so relocates the entry. This is simple but reads every entry from disk. *Index-based relocation* iterates through Large Table cells, loads each index, identifies entries whose WAL positions fall below a configurable cutoff, reads their values, and decides which to relocate. This is more efficient when only a subset of cells contain old entries. Both strategies support an optional application-provided filter that decides for each entry: keep, remove, or stop relocation. This enables application-specific compaction logic, such as retaining only the latest version of each key. An important use is *epoch-based garbage collection*, the pattern used by Sui and other systems that organize data into time windows or versions. The application groups entries by epoch and maintains a retention boundary (the oldest epoch it must still serve); older entries can be discarded. The filter returns `Remove` if the entry's epoch is below the boundary, otherwise `StopRelocation`. Because epochs commit in order, the WAL is partitioned into contiguous epoch-regions in append order: once the scan reaches a still-retained epoch, every later entry is too. `StopRelocation` ends the pass at exactly this boundary, without reading the rest of the WAL.

A key architectural difference from LSM-tree databases is the role of compaction. In LSM-trees, compaction is essential for read performance: without it, reads search multiple SSTable levels and read amplification grows with level count. In Tidehunter, read performance is independent of WAL fragmentation: the Large Table index provides constant-cost access to any value (one index lookup plus one WAL read) regardless of where in the WAL it lives. Relocation thus serves a single purpose: reclaiming disk space. Each value is written to the WAL only once during normal operation; additional writes occur only when relocation runs, and even then only on live entries. Where LSM-tree compaction trades write amplification for read performance, Tidehunter's relocation trades it purely for disk space, and applications with ample storage can defer or skip it without affecting read latency.

5 IMPLEMENTATION

Our Tidehunter implementation is *production-ready*, written in Rust [35] (about 14,000 lines, excluding tests), and being deployed to the Sui blockchain. Key libraries are: *memmap2* for memory-mapped I/O, enabling direct WAL-fragment writes without explicit system calls; *parking_lot* for high-performance mutexes with Arc-based guards used in position tracking; *arc-swap* for atomic Arc-pointer swaps enabling lock-free reads of shared structures like the WAL's memory-map registry (its set of active memory-mapped regions); *crc32fast* for WAL-entry checksums; *bloom* for per-cell Bloom filters; *lru* for the value cache; and *rayon* for parallel Large Table initialization during recovery. Unsafe code is confined to memory-mapped I/O within the WAL module. The codebase includes 160 unit tests, stress tests with shadow-state verification, and a failpoint infrastructure for deterministic concurrency testing.

Tidehunter uses a fixed set of background threads. Each WAL Controller (Value and Index) runs three: a *tracker* monitoring position completion, a *mapper* managing memory-map lifecycle and GC, and a *syncer* issuing fsync on finalized fragments. A configurable pool of *flusher* threads handles index persistence concurrently across cells. A *relocator* thread performs background compaction, and a *snapshot* thread periodically checkpoints the Large Table. A typical deployment runs 8 background threads plus the configured flushers; we deliberately avoid per-request thread spawning to keep scheduling overhead and resource usage predictable.

These background threads minimize interference with client requests. Foreground threads dispatch work (index flushes, fsync, map finalization, relocation) to background threads via non-blocking message queues and return immediately. Row-level locks are held only for brief in-memory index updates and are released before any I/O or cross-thread handoff. Background threads operate on reference-counted snapshots and use atomic pointer swaps so readers see a consistent view without coordination. The relocator updates the index via compare-and-set against the last-processed position (§4.4) rather than holding cell locks, running fully concurrently with foreground writes. To bound memory when index persistence falls behind ingestion, the flusher applies back-pressure: when the number of in-flight index flushes exceeds a threshold, the foreground thread requesting the flush sleeps briefly (outside any row lock) until pending flushes drain, trading a small throughput dip for bounded memory.

6 EXPERIMENTAL EVALUATION

6.1 Experimental Setup

We run our experiments on a Medium v4 OpenMetal instance [38] equipped with two Intel Xeon Silver 4510 processors providing 24 CPU cores (48 threads) at 2.4–4.1 GHz, 256 GB of DDR5 4400 MHz RAM, and a 6.4 TB NVMe drive.

Baselines. We compare Tidehunter against RocksDB [32], a widely deployed LSM-tree store, and BlobDB [30], RocksDB's large-value extension inspired by WiscKey [29].

Resource Parity and Configuration. We allocate equivalent resources to all systems to ensure a fair comparison. All three systems use the same client threads (36 write threads and 36 mixed read/write threads) and the same number of background threads: TIDEHUNTER uses 12 flusher threads, while RocksDB and BlobDB are configured with `increase_parallelism(12)`. For memory, TIDEHUNTER uses up to 128 GiB of memory-mapped WAL in total (64 fragments of 1 GiB each per WAL, applied to both the Value WAL and the Index Store), whose physical residency is managed by the OS page cache. RocksDB and BlobDB use a 128 MB block cache plus the OS page cache for SST files. On our 256 GB machine, both designs therefore rely on the OS page cache as a cache layer. We verified that increasing RocksDB's block cache to 128 GB with direct reads (to avoid double-caching) degrades throughput by 5–16% across all workloads, confirming that our smaller-cache configuration is already favorable to RocksDB. Our RocksDB compression (LZ4 with ZSTD at the bottommost level), 16 KB block size, 10-bit Bloom filters, and pinned L0 index/filter blocks follow Meta's RocksDB tuning recommendations [33], and our BlobDB settings follow the BlobDB guidance [31]. Remaining values (write buffer sizes, parallelism, L0 triggers) were selected by tuning on our 1 TB workload. Finally, all three systems use background fsync (`sync=false` for RocksDB/BlobDB, `sync_flush: false` for TIDEHUNTER), matching RocksDB's default.

Methodology. Our benchmark has two phases. In the *fill phase*, we insert random keys and values until the database reaches 1 TB (unless specified otherwise); we use a fixed 32-byte key, and the value size is a parameter. §6.2.3 additionally evaluates smaller key/value combinations drawn from production workload studies. After fill, we allow a 10-minute cooldown for the system to stabilize (e.g., for relocation to settle). The *measurement phase* then runs operations for 10 minutes. Following YCSB's convention [12], lookups always target keys present in the database, so reads do not produce negative lookups. We sweep five workload parameters: read-write ratio (100% writes, 50/50, or 100% reads); read operation type (*get*, *exists*, or reverse iterator returning the largest key smaller than the query); access skew (Zipfian $\theta = 0$ homogeneous or $\theta = 2$ favoring recently inserted keys); value size (1 KB, 128 B, or 64 B); and relocation (enabled by default).

6.2 Benchmark Results

We present benchmark results for TIDEHUNTER, RocksDB, and BlobDB across various workload configurations.

6.2.1 *Large Values.*

The results for large values are in Figure 6.

Write Performance. Under write-only workloads (Figure 6a, leftmost bars), TIDEHUNTER outperforms RocksDB by 6.8× and BlobDB by 2.2×. This advantage stems from TIDEHUNTER's append-only WAL design: each write requires only a single sequential disk operation, whereas LSM-tree-based systems must eventually compact and rewrite data multiple times. BlobDB partially mitigates this overhead by separating large values from the LSM-tree, explaining its intermediate performance.

Read Performance. For homogeneous reads, TIDEHUNTER maintains a consistent advantage across all operation types. On get operations (Figure 6a), TIDEHUNTER outperforms RocksDB by 2.0×; on exists, by 21.1×, because TIDEHUNTER resolves existence queries directly from the index without fetching the value, while LSM-trees must traverse multiple levels regardless of whether the value is needed. At this dataset size, TIDEHUNTER's ≈48 GB on-disk index (1.1 B records × 44 B/entry) fits entirely within the 64 GiB Index Store mmap window, so exists is a pure in-memory Bloom-filter plus index lookup; on a larger dataset that evicts the index, it would fall back to the optimistic index's 1–3-page disk-read path (§4.2). For reverse iterator operations (Figure 6a), TIDEHUNTER maintains a 3.5× advantage over RocksDB.

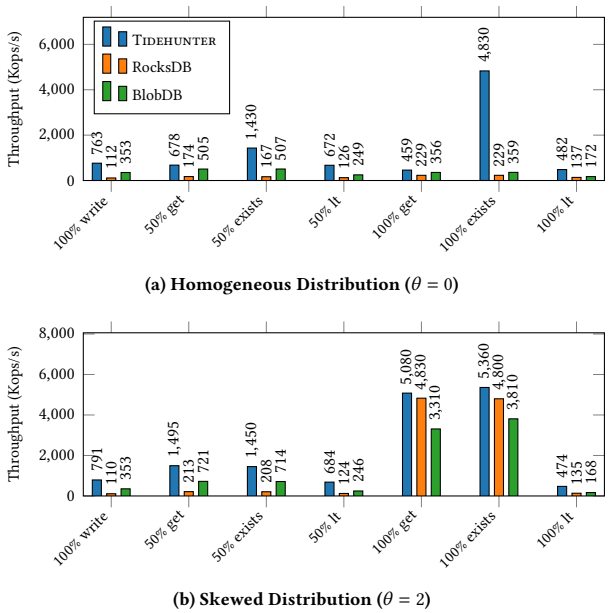


Figure 6: Throughput comparison across workloads. The 50% read cases include 50% writes. Top: homogeneous access distribution ($\theta = 0$). Bottom: skewed distribution ($\theta = 2$).

Mixed Workloads. At a 50/50 read-write ratio, TIDEHUNTER leads RocksDB by 3.9× on get operations. The gap narrows relative to write-only workloads but remains substantial, showing that TIDEHUNTER's design also benefits mixed read-write paths.

Effect of Skew. Under skewed workloads (Figure 6b), where accesses favor recently inserted keys, all systems benefit from caching effects. For read-only get operations with skew, TIDEHUNTER and RocksDB land within 5% of each other, with neither backend holding a meaningful lead. Hot data resides in memory across all systems: RocksDB's block cache and TIDEHUNTER's Large Table both serve frequently accessed entries without disk I/O, and the two designs converge on this workload. However, TIDEHUNTER maintains its advantage for write-heavy and mixed workloads even under skew, as its write path remains fundamentally more efficient.

Performance Stability. To demonstrate that TIDEHUNTER's background threads do not meaningfully interfere with foreground operations, we ran 6 workload configurations (3 read ratios × 2 skew

Table 1: Stability across 6 workload configurations on 1 KB values. Throughput CV is the worse of write- and read-throughput CVs. "LT ovhd" is the expected per-operation wait on Large Table row mutexes, reported as a fraction of mean operation latency.

			Latency p50/p99/p999 (μs)		
R%	θ	Tput CV	Write	Read	LT ovhd %
0	0.0	0.07	28/68/234	–	0.0004
50	0.0	0.12	9/37/67	93/465/1196	0.0001
100	0.0	0.07	–	86/538/1292	<0.0001
0	2.0	0.07	28/69/270	–	0.0004
50	2.0	0.08	24/58/193	3/20/140	0.0010
100	2.0	0.006	–	4/15/19	0.068

settings) for 30 min each on a single node with 1 KB values; Table 1 reports the results. Three observations establish the claim. First, foreground tail latency is tightly bounded even while the flusher and relocator run continuously: write p999 stays under 0.3 ms and read p999 under 1.3 ms across every configuration; background work does not stall foreground operations. Second, foreground lock contention on Large Table row mutexes is negligible: in five of six configurations it accounts for at most 0.001% of mean operation latency. The only outlier (100% reads, $\theta = 2$) reaches 0.07%, and even that reflects foreground readers colliding on the small set of hot cells rather than background activity; TIDEHUNTER's background threads are not the source even when contention is visible in the data. Third, throughput is steady: the coefficient of variation stays at or below 0.12 in every configuration. This variance comes from TIDEHUNTER's flusher back-pressure mechanism (§5) which briefly throttles foreground writes when pending flushes saturate.

6.2.2 Small Values. The results for small values are in Figure 7. For small values, the performance characteristics differ substantially from the large-value case, revealing the trade-offs inherent in TIDEHUNTER's design.

Write Performance. Under write-only workloads (Figure 7a, leftmost bars), RocksDB and BlobDB outperform TIDEHUNTER by 2.2×. This reversal occurs because the overhead of TIDEHUNTER's separate index becomes proportionally more significant when values are small. LSM-trees amortize their metadata costs across sorted runs and benefit from efficient compression of collocated entries.

Read Performance. The gap narrows for read operations under homogeneous access patterns. For get operations at 100% reads (Figure 7a, rightmost bars), RocksDB holds a 1.4× advantage. This residual gap reflects LSM-trees' cache efficiency advantage: sorted runs pack many small entries into each cached block, whereas TIDEHUNTER's position-based lookups touch one cached page per lookup. Notably, TIDEHUNTER's exists operation (Figure 7a) holds a 1.3× advantage over RocksDB at 100% reads, showing that our existence-check optimization provides value even for small entries.

Effect of Skew. Skewed workloads (Figure 7b) reveal an important exception to LSM-trees' small-value advantage. Under read-only workloads with high skew, TIDEHUNTER surpasses RocksDB by

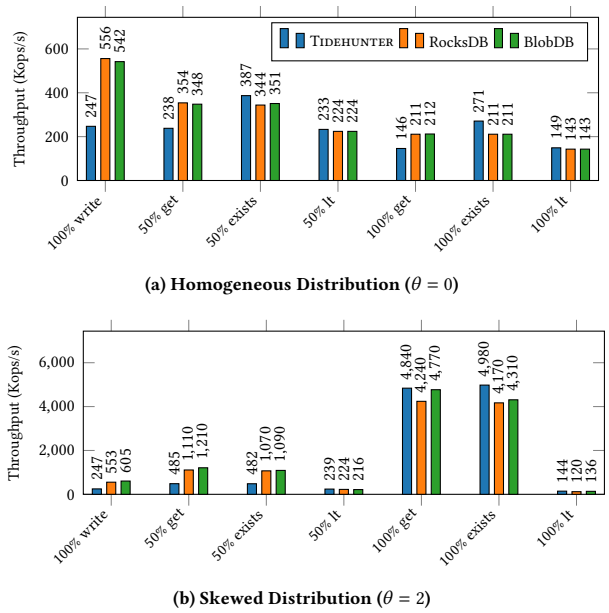


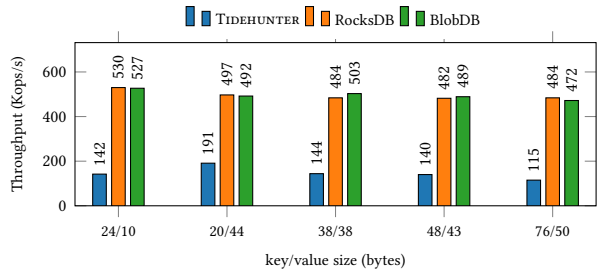
Figure 7: Throughput comparison for small values (64 bytes). Top: homogeneous access distribution ($\theta = 0$). Bottom: skewed distribution ($\theta = 2$).

14% on get and 19% on exists. This reversal occurs because TIDEHUNTER's Large Table efficiently caches hot entries, and the WAL's append-only structure means recently written (and frequently accessed under skew) entries are contiguous, improving cache locality.

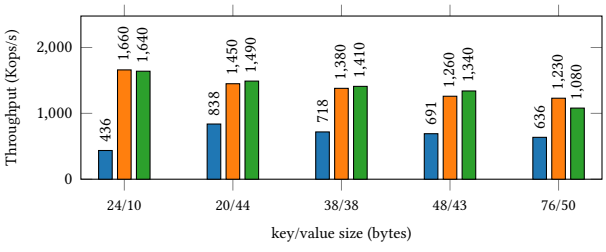
We also evaluate 128-byte values, an intermediate point between the 64-byte and 1 KB regimes; results are in the full version [11].

6.2.3 Application-Workload Regimes. The previous subsections vary value size with a fixed 32-byte key. To characterize TIDEHUNTER's boundary on workloads with different shapes, we evaluate five key/value combinations drawn from production-trace studies of small-record key-value workloads at hyperscalers [41]. The combinations are (key/value, in bytes): (24/10) corresponding to RTDATA, (48/43) corresponding to ZippyDB, and three intermediate points (20/44), (38/38), and (76/50). These regimes have small entries dominated by key bytes, the opposite of TIDEHUNTER's favored large-value regime. They probe the small-record edge of the design rather than its core target. We pre-fill 500 GB of raw user bytes per backend and run a 30-minute measurement phase with 50% reads and 50% writes, under both homogeneous ($\theta = 0$) and skewed ($\theta = 2$) access. Results are in Figure 8.

TIDEHUNTER regresses on small-record mixed workloads. Across all five combinations under homogeneous access (Figure 8a), TIDEHUNTER's throughput sits 2–4× below RocksDB and BlobDB. TIDEHUNTER's per-record WAL and index metadata are constant-size and therefore dominate when the entry itself is only 30–90 bytes; LSM-trees pack many such entries into each cached block and amortize their metadata cost over them.



(a) Homogeneous Distribution ($\theta = 0$)



(b) Skewed Distribution ($\theta = 2$)

Figure 8: Throughput on five small-record application workloads with 50% reads / 50% writes. K/V combinations are listed as "key/value" bytes, ordered by total entry size. (24/10) and (48/43) match the RTDATA and ZippyDB workload signatures from [41]; the other three are intermediate points. Top: homogeneous ($\theta = 0$). Bottom: skewed ($\theta = 2$).

The regression persists under skew on the 50/50 mix. Unlike the 64-byte case where skew lets TIDEHUNTER overtake LSM-trees for read-only get operations (Figure 7b), at these regimes RocksDB and BlobDB remain ahead under $\theta = 2$ across all five combinations (Figure 8b). Skew helps every backend, but the LSM block cache holds far more hot small records than TIDEHUNTER's Large Table at this scale, and the existing 50% writes prevent skew from collapsing into a pure cache-hit microbenchmark.

Summary. These results delineate TIDEHUNTER's deployment boundary. TIDEHUNTER is engineered for workloads where values are large enough that per-record metadata is amortized cheaply and where LSM-tree compaction is the dominant cost (content-addressable storage, deduplication, and the blockchain-validator workload that motivated its design). Workloads with key-dominated small records, such as those characterized by [41], fall outside this regime; for those, LSM-tree designs remain the appropriate choice.

Effect of Relocation. To measure relocation's storage and performance impact, we ran a 100%-delete workload under uniform (θ=0) and skewed (θ=2) access on a 1 TB pre-fill of 1 KB values (plots in the full version [11]). Without relocation, deleted entries leave gaps that accumulate to roughly the original 1 TB under both patterns. With relocation, storage drops by 71% (uniform) and 19% (skewed); the smaller savings under skew reflect that only frequently accessed keys are deleted, leaving fewer reclaimable gaps. Throughput overhead is minimal: 3% uniform, 4% skewed.

6.2.4 Write Amplification and Tail Latency Under Sustained Churn. We evaluate TIDEHUNTER's foreground behavior under workloads that continuously update or delete keys, asking: (i) what write amplification (WA) does the application observe at steady state, (ii) how much does enabling relocation cost in foreground throughput and tail latency, and (iii) can the relocation reclaim threshold be tuned without imposing a foreground tax?

Each configuration pre-fills 500 GB of (32 B key, 1 KB value) data with pure inserts, then runs a 60-minute pure-write measurement phase. Measurement operations split into overwrites, deletes, and fresh inserts; the mix is controlled by the (overwrite, delete) ratio. We compute WA as bytes written to the data device divided by application-visible bytes during measurement.

Strategy comparison (Table 2). We swept three relocation strategies (None, WalBased, IndexBased) across three workload mixes. Within each workload, WA is invariant across strategies to within ±0.02 and foreground throughput differs by less than 2%. Enabling relocation does not amplify writes: the rewrite cost runs on background threads with no read-path coupling and does not appear in the application's I/O accounting. The absolute WA of ≈2.0 in the 100% delete row is not caused by relocation (the no-relocation baseline shows the same value). Instead, it reflects the fixed per-record WAL metadata overhead: each delete writes only a ≈32 B tombstone as application data, so the constant metadata bytes are large relative to that payload, roughly doubling the ratio.

Cross-system contrast (Table 2, BlobDB rows). We rerun this sweep with BlobDB as a cross-system reference point on the same hardware, workload, and resource budget. BlobDB's compaction-driven value-log GC pays for sustained churn on every axis. Foreground throughput drops to 42–67% of TIDEHUNTER's across the three workloads. Write-path p99.9 latency inflates 7–43×, because of compaction threads sharing the foreground I/O path: most operations land cheaply in the memtable, while periodic compactions stall a small fraction of operations for milliseconds. WA is also higher in every workload: roughly 30% higher on overwrite-heavy mixes and nearly 3× on pure deletes. Each BlobDB delete propagates a tombstone through every LSM level via compaction and the blob file holding the deleted value must eventually be rewritten to drop the dead entry, cascading writes that dominate disk I/O even though the application's per-delete intent is only a 32 B key. TIDEHUNTER instead appends a 32 B tombstone and updates the in-memory index; the relocator is decoupled from the foreground.

Threshold sweep. We sweep the relocation percentage (the bound on the fraction of WAL a single relocation pass may rewrite) over {1, 5, 10, 25, 50}% on the 50/50 mixed workload with WalBased relocation. WA stays flat at 1.19–1.20, throughput within 2%, and tail latency at p99 and p999 within 6% across the entire range. Operators can dial reclaim aggressiveness without paying a foreground tax.

6.2.5 Recovery. We measure how long TIDEHUNTER takes to come back online after a restart, varying database size, snapshot cadence, and whether the shutdown was clean (Table 3). Each row corresponds to one restart on a pre-filled database; we report wall-clock recovery time and a first-read latency from 1,000 single-threaded random reads issued immediately after open.

Table 2: Foreground behavior under sustained measurement-phase churn. *WA*: bytes written to data device divided by application bytes written, measurement phase only. *Tput*: foreground operations per second. *p99/p999*: write-path latency in μs.

Workload	Strategy	WA	Tput	p99/p999
100% overwrite	None	1.14	857 K	62/76
	WalBased	1.15	853 K	61/76
	IndexBased	1.16	843 K	60/76
	BlobDB	1.54	361 K	2,190/3,279
50/50 mix	None	1.19	992 K	89/113
	WalBased	1.20	998 K	88/111
	IndexBased	1.22	992 K	88/112
	BlobDB	1.51	662 K	78/2,184
100% delete	None	2.12	2,070 K	115/157
	WalBased	1.98	2,080 K	116/159
	IndexBased	2.00	2,100 K	116/160
	BlobDB	5.82	959 K	51/1,104

Table 3: Recovery measurements. "Repl." shows bytes processed during recovery. "Total" is wall-clock recovery time. "First-read" percentiles are over 1,000 random keys read. In the snapshot-interval block, ∞ disables periodic snapshots.

	DB (GB)	Snap (GB)	Repl. (GB)	Total (s)	First-read (μs) p50/p99
Cold-start vs DB size, 128 GB snapshot					
cold-100	100	128	109	180	4/7
cold-500	500	128	140	245	81/350
cold-1tb	1,024	128	245	465	352/995
Snapshot interval at 1 TB					
snap-16	1,024	16	263	432	299/945
snap-64	1,024	64	243	401	312/922
snap-128	1,024	128	207	350	289/985
snap-256	1,024	256	317	566	307/977
snap-∞	1,024	–	1,116	1,791	127/1,328
200 GB fill, crash during relocation					
crash	200	128	85	146	82/232

Recovery time is essentially Value WAL replay. The other phases (acquiring the lock, reading the Control Region, opening the Value WAL, initializing the Large Table, opening the Index Store writer, and starting background threads) together take under 0.5% of the total in every run.

Cold-start scaling. Recovery time grows much more slowly than the database: a 10× size increase from 100 GB to 1 TB translates into only a 2.6× increase in recovery time. The amount of WAL replayed has a floor set by the snapshot interval, which is why the 100 GB and 500 GB rows replay roughly the same. Only the 1 TB row replays substantially more, because more writes had accumulated past the most recent snapshot. First-read p99 latency rises sharply with database size: the larger on-disk Index Store means each random read touches more cold pages.

Worst-case snapshot cadence. The biggest recovery lever is turning periodic snapshots on at all: a 1 TB database without snapshots must replay the entire WAL (nearly 30 min), while any finite cadence cuts this by 3–5×. Within finite cadences, smaller is not always faster, because a Tidehunter snapshot writes only a Control Region pointer; it does not synchronously flush dirty cells from the Large Table. That pointer can advance no further than the WAL offset of the oldest dirty cell, so under sustained writes the recovery start point lags the WAL head by much more than the snapshot interval. The practical takeaway: enable periodic snapshots; tuning the cadence yields only modest additional gains.

Crash during relocation. We also checked that an unclean shutdown during relocation leaves the database no worse than a clean one. We pre-filled 200 GB, ran continuous Value WAL relocation in the background, and killed the process abruptly: no destructors ran and no final fsync was issued, so only data already in the OS page cache survived. Recovery completes promptly (Table 3); first-read latency is in line with cold-start results, showing that continuous relocation does not degrade post-crash read performance.

6.2.6 Memory Footprint and Sensitivity. We quantify RAM usage analytically (per-key and per-cell metadata), via runtime instrumentation of the main eval point (hit ratios and eviction behavior), and via sensitivity sweeps over the parameters governing the memory–performance trade-off.

Per-key and per-cell footprint. The Large Table stores each key as a fixed-format entry in its cell's sorted in-memory index, which uses the same layout as the on-disk index (§4): key size +12 B per entry (8 B Value WAL offset + 4 B length word), so 32 B keys cost 44 B per resident entry. Per cell, the always-resident metadata is approximately 440 B: a 400 B per-cell metadata block (cell identifier, a handle to the on-disk index, the pending-update table, four metric-delta trackers, optional Bloom-filter and value-cache slots) plus a 40 B row-level wrapper holding the cell mutex and the entries array. With the default 2^{17} cells, cell metadata totals about 55 MiB. Dirty-but-unflushed entries are buffered in a per-cell write buffer whose size is bounded by the per-cell dirty-key cap (`max_dirty_keys`), which defaults to 1024 entries per cell; at ~80 B per entry (32 B key + 16 B value + BTreeMap node overhead), this buffer is bounded at ~80 KiB per cell.

Cache layer and measured hit ratio. Tidehunter's only configured value cache is the memory-mapped Value WAL (128 GiB window on a 256 GB machine, with physical residency managed by the OS page cache); the optional LRU is disabled in all paper experiments. We instrumented Tidehunter's runtime metrics on the main eval point (1 TB pre-fill, 32 B keys, 1 KB values, 50/50 mixed Get, $\theta = 0$, 30 min).

Of ~600 M lookups, 4.3% resolved in the Large Table in-memory index and 95.7% required the on-disk Index Store; the LRU and Bloom sources accounted for 0%, because the LRU is disabled, and our read workload follows YCSB's positive-lookup convention (§6.1). Under this uniform workload over $\sim 10^9$ keys, the Large Table's role is not to serve reads directly but to (a) write-batch new keys until the per-cell dirty-key cap is exceeded and the cell is

flushed to the Index Store, and (b) hold the position estimator that resolves on-disk lookups in ≤ 2 disk reads (§6.3).

Eviction dynamics. The measurement phase issued 694 K flushes (~370/s), which flushed 7.1 B keys and churned 313 GB of index over 30 minutes (~170 MB/s of Index Store writes). Flush backpressure (§5) triggered ~3 M times, the dominant source of the foreground throughput variance in Table 1, and the snapshot force-relocated 7.3 K cells whose on-disk index lay below the relocation cutoff. At the end of the phase, Tidehunter held 137.4 GB of mmap'd WAL and 6.44 GB of estimated total memory: 4.0 GB of resident key bytes, 3.7 GB of sorted in-memory index, and 82 M dirty keys awaiting flush. These in-memory totals oscillate between 1.4 GB and 6.2 GB, since the end-of-phase sample can land anywhere in the flush–accumulate cycle; the mean working set is ≈4 GB.

Sensitivity to Bloom-filter FPR. We swept the Bloom target false-positive rate p over four points from 0.001 to 0.1, at a fixed workload (1 KB values, 32 B keys, 100% Get, $\theta = 0$, 1 TB pre-fill). Because our read workload follows YCSB's positive-lookup convention (§6.1), every Get queries a key that is present in the database, so the Bloom always returns a true positive regardless of p. Foreground throughput is therefore essentially flat across the sweep (456–470 K ops/s, <3% spread), so for this workload the parameter trades only memory. The Bloom array size per cell is $\left(\lceil -n \ln p/(\ln 2)^2 \rceil\right)$ bits, where n is the per-cell Bloom capacity in keys (`bloom_filter_count`). With the default $n = 8192$, this falls from 14.4 bits per key at $p = 0.001$ to 4.8 at $p = 0.1$. Summed over the default 2^{17} cells, the total Bloom footprint at $p = 0.01$ is ~1.3 GiB.

Sensitivity to cell sizing and dirty-key thresholds. We swept the two memory-shaping parameters, cell count (`num_mutexes`) and per-cell dirty-key cap (`max_dirty_keys`), on the standard 50/50 / $\theta = 0$ / 1 KB workload (Table 4). Both axes validate the production defaults (2^{17} cells, cap 1024), with throughput falling away on either side: by 30% at 2^{14} cells (each pending-write merge churns an 8× larger sorted array) and 22% at 2^{20} (per-cell metadata pressure), and by 49% at a 64-entry cap, where premature unloads also blow out p999 to 14 ms; raising the cap beyond the default yields no measurable gain. Tail latency tracks throughput on both axes.

Sensitivity to mmap window size. Sweeping the per-WAL fragment count (`max_maps`) at a fixed 1 GiB fragment size traces the mmap-window curve in Table 4: throughput rises from 640 K ops/s at a 32 GiB total window to a peak of 677 K at 128 GiB, then falls to 612 K at 256 GiB, where the mmap'd address space approaches the machine's 256 GB of RAM and the kernel loses headroom for write buffers and non-WAL state; p99.9 climbs from ~290 μs to 896 μs across the same range. The 128 GiB setting (64 fragments per WAL) is both the production configuration and the throughput-optimal choice on this hardware, and we use it throughout the paper.

6.3 Index Format Comparison

We microbenchmark the lookup performance of Tidehunter's two persistent index formats: the optimistic index and a naive baseline. The baseline is a header-based index that uses a fixed 128-entry header to partition keys into micro-cells, always performing exactly two I/Os per lookup; the optimistic index (§4.2) instead estimates each key's position from the uniform key distribution and binary-searches a small window around it. On 25,000 sorted index files (1 M entries each, 32-byte keys, 1 TB total, predominantly negative-lookup workload), the optimistic index outperforms the header-based baseline by 24% at 48 threads. Full sensitivity sweeps over window size and thread count are in the full version [11].

Table 4: Memory-sensitivity sweeps.

Parameter	Value	Tput	p99/p999 (μs)
Total mmap window (across both WALs)			
window	32 GiB	640 K	201 / 285
	64 GiB	660 K	199 / 280
	128 GiB (default)	677 K	207 / 469
	256 GiB	612 K	311 / 896
Cell count (one mutex per cell)			
cells	2^{14} (16,384)	429 K	409 / 1,276
	2^{16} (65,536)	591 K	333 / 830
	2^{17} (131,072, default)	612 K	312 / 878
	2^{19} (524,288)	581 K	365 / 901
	2^{20} (1,048,576)	478 K	593 / 1,381
Per-cell dirty-key cap			
cap	64	312 K	653 / 14,422
	256	553 K	386 / 934
	1,024 (default)	612 K	312 / 878
	4,096	588 K	352 / 1,012
	16,384	603 K	324 / 934

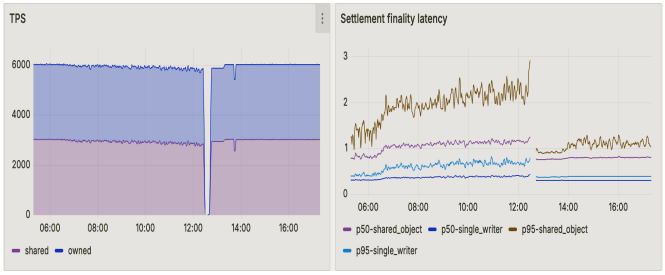
6.4 Case Study: Sui Blockchain

To demonstrate real-world performance, we integrated Tidehunter into the Sui blockchain, which currently uses RocksDB as its database backend. We replaced Sui's validator storage layer with Tidehunter and deployed in a cluster of 150 geographically distributed machines under an artificial load of 6,000 transactions per second.

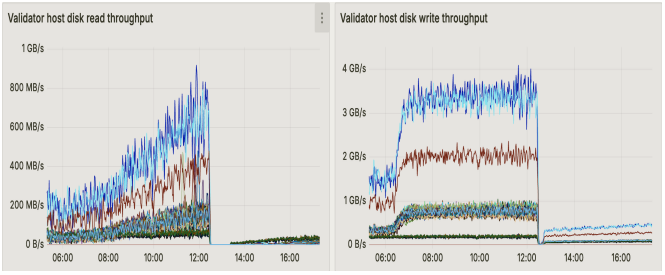
The hosts started the experiment using RocksDB and switched to using Tidehunter. These results are presented in Figures 9a and 9b. The first thing to notice is that under the sustained load of 6,000 TPS, RocksDB performance degrades after several hours, resulting in significant latency increases (Figure 9a before 12:30). In contrast, once Tidehunter is deployed the same hosts maintain stable latency throughout the test period (Figure 9a, after 12:30). The same characteristics are also present in the disk I/O utilization (Figure 9b), where the high write amplification of RocksDB results in 1–4 GB/s write throughput demand, whereas the need to search deep inside the LSM-Tree results in 0.2–0.8 GB/s read throughput demand. In contrast, once Tidehunter is enabled the demand of the disk resource drops to less than 100 MB/s.

Since Tidehunter sustained 6,000 TPS without any issue, we ran a second experiment with a load of 8,500 TPS; throughput and latency remain steady during five hours of sustained load (plot in the full version [11]).

As of January 2026, Tidehunter is deployed on Sui's devnet, and several testnet full nodes have been permanently migrated from RocksDB to Tidehunter.



(a) Transaction latency. RocksDB (before 12:30) degrades over time; TIDEHUNTER (after 12:30) stays stable.



(b) Disk I/O. TIDEHUNTER (after 12:30) achieves significantly lower disk read and write rates.

Figure 9: Sui blockchain under 6,000 TPS load: the same 150 validators run RocksDB before 12:30 and TIDEHUNTER after.

7 RELATED WORK

LSM-Tree Stores. LSM-trees [37] organize data into sorted runs across multiple levels, trading write amplification for read performance through background compaction. Stores like RocksDB [8, 20], LevelDB [16], and PebblesDB [43] have become industry standards, with various optimizations including lazy leveling [14], optimal Bloom filter allocation [13], and concurrent compaction [21]. Despite these improvements, compaction fundamentally rewrites data 5–10× across levels. TIDEHUNTER sidesteps this entirely: its append-only WAL eliminates multi-level compaction, achieving 6.8× higher write throughput than RocksDB.

Key-Value Separation. WiscKey [29] pioneered storing values in a separate log (vLog), dramatically reducing write amplification for large values. Production successors include BlobDB [30], Titan [42], DiffKV [28], HashKV [9], BVLSM [27], Kreon [40], and BadgerDB [18], which variously optimize value placement, garbage collection, and flash-friendliness. Most keep an LSM-tree for keys, inheriting its compaction overhead: key compaction still causes write stalls, multi-level lookups still suffer under uniform key distributions, and vLog garbage collection contends with foreground writes. TIDEHUNTER instead makes the WAL the unified value store with lazy-flushed index tables, avoiding both LSM compaction and separate vLog management.

B-Tree and Memory-Mapped Stores. LMDB [10] uses copy-on-write B+ trees with shadow paging, enabling zero-copy reads but limited to single-writer concurrency. CedrusDB [47] combines memory-mapped lazy-tries with a WAL and is the most architecturally similar prior work to TIDEHUNTER. However, CedrusDB's in-place trie updates require per-node locking during traversal, limiting write concurrency to approximately eight threads; its read path must walk the trie from root to leaf regardless of key distribution. TIDEHUNTER's append-only WAL with atomic position allocation sustains high concurrent write throughput, and its optimistic index bypasses tree traversal entirely for on-disk lookups. Traditional B-tree systems with logging and locking include BerkeleyDB [36] and redb [6]. Unlike these, TIDEHUNTER's append-only design avoids page splits and tree rebalancing entirely.

Log-as-Database Systems. Several systems share TIDEHUNTER's philosophy of treating the log as the primary data store, but at different abstraction levels and for different bottlenecks. Amazon Aurora [46] sends only redo log records over the network; its storage tier materializes pages from redo records, reducing *network* amplification in a distributed RDBMS. ERMIA [23] replaces in-memory heap pages with log-structured version records and OID-array indirection, targeting MVCC efficiency in DRAM. LLAMA [26] provides a latch-free, log-structured cache/storage substrate for access methods like the Bw-Tree [25], using delta chains and CAS-based page updates; it relies on cleaning and consolidation both to reclaim space and to shorten delta chains, which also improves read performance. In contrast, TIDEHUNTER targets *disk* write amplification for large-value key-value workloads: its WAL permanently stores values (not redo records or version deltas), and its relocation is optional because read performance is independent of WAL fragmentation.

Update-in-Place and Learned Indexes. TreeLine [48] also rejects LSM compaction on modern NVMe SSDs, but uses update-in-place B-tree pages with record caching and model-guided page grouping, embracing random writes where SSDs' sequential-to-random gap has narrowed. TIDEHUNTER takes the opposite approach: strictly sequential, append-only writes that achieve minimal write amplification while trading away sorted iteration. Kraska et al. [24] observe that indexes are models of the data distribution's CDF and show that learned models (from linear regression to neural networks) can replace traditional index structures. TIDEHUNTER's optimistic index shares this CDF-as-position-estimator insight but exploits the uniform key distribution directly, where the CDF is a trivial linear function, requiring no training or model maintenance and naturally supporting dynamic data.

Write-Ahead Logging. ARIES [34] established foundational WAL protocols with steal and no-force buffer management for B-tree databases. Silo [45] uses epoch-based transaction processing with group commit, which inspired TIDEHUNTER's batched WAL syncing. Hekaton [19] optimizes logging for in-memory OLTP, while Write-Behind Logging [2] targets persistent memory. Unlike traditional systems where WAL is auxiliary for recovery, TIDEHUNTER's WAL is the primary data store with lazy index flushing.

REFERENCES

[1] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. 2009. FAWN: A Fast Array of Wimpy Nodes. In *SOSP*. ACM, Big Sky, MT, 1–14.

[2] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-Behind Logging. *Proceedings of the VLDB Endowment* 10, 4 (2016), 337–348.

[3] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *USENIX ATC*. USENIX Association, Renton, WA, 753–766.

[4] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. 2010. Finding a Needle in Haystack: Facebook's Photo Storage. In *OSDI*. USENIX Association, Vancouver, Canada, 47–60.

[5] Juan Benet. 2014. IPFS - Content Addressed, Versioned, P2P File System. arXiv:1407.3561.

[6] Christopher Berner. 2023. redb: An Embedded Key-Value Database in Pure Rust. Retrieved July 27, 2026 from https://github.com/cberner/redb

[7] Sam Blackshear, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Xun Li, Mark Logan, Ashok Menon, Todd Nowacki, Alberto Sonnino, Brandon Williams, and Lu Zhang. 2024. Sui Lutris: A Blockchain Combining Broadcast and Consensus. In *CCS*. ACM, Salt Lake City, UT, 2606–2620. https://doi.org/10.1145/3658644.3670286

[8] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *FAST*. USENIX Association, Santa Clara, CA, 209–223.

[9] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. 2018. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *USENIX ATC*. USENIX Association, Boston, MA, 1007–1019.

[10] Howard Chu. 2011. MDB: A Memory-Mapped Database and Backend for OpenLDAP. In *LDAPCon*. LDAPCon, Heidelberg, Germany, 1–10.

[11] Andrey Chursin, Lefteris Kokoris-Kogias, Alex Orlov, Alberto Sonnino, and Igor Zablotchi. 2026. Tidehunter: Large-Value Storage With Minimal Data Relocation. arXiv:2602.01873.

[12] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*. ACM, Indianapolis, IN, 143–154. https://doi.org/10.1145/1807128.1807152

[13] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *SIGMOD*. ACM, Chicago, IL, 79–94.

[14] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *SIGMOD*. ACM, Houston, TX, 505–520.

[15] Javier de la Rúa Martínez, Fabio Buso, Antonios Kouzoupis, Alexandru A. Ormenisan, Salman Niazi, Davit Bzhalava, Kenneth Mak, Victor Jouffrey, Mikael Ronström, Raymond Cunningham, Ralfs Zangis, Dhananjay Mukhedkar, Ayushman Khazanchi, Vladimir Vlassov, and Jim Dowling. 2024. The Hopsworks Feature Store for Machine Learning. In *SIGMOD Companion*. ACM, Santiago, Chile, 135–147.

[16] Jeffrey Dean and Sanjay Ghemawat. 2011. LevelDB. Retrieved July 27, 2026 from https://github.com/google/leveldb

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT*. Association for Computational Linguistics, Minneapolis, MN, 4171–4186.

[18] Dgraph Labs. 2017. BadgerDB: Fast Key-Value DB in Go. Retrieved July 27, 2026 from https://github.com/dgraph-io/badger

[19] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *SIGMOD*. ACM, New York, NY, 1243–1254.

[20] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. RocksDB: Evolution of Development Priorities in a Key-Value Store Serving Large-Scale Applications. *ACM Transactions on Storage* 17, 4 (2021), 1–32.

[21] Robert Escriva. 2013. HyperLevelDB: A Fork of LevelDB Optimized for Concurrency. Retrieved July 27, 2026 from https://github.com/rescrv/HyperLevelDB

[22] Fanglu Guo and Petros Efstathopoulos. 2011. Building a High-performance Deduplication System. In *USENIX ATC*. USENIX Association, Portland, OR, 285–298.

[23] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *SIGMOD*. ACM, San Francisco, CA, 1675–1687.

[24] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *SIGMOD*. ACM, Houston, TX, 489–504.

[25] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for New Hardware Platforms. In *ICDE*. IEEE, Brisbane, Australia, 302–313.

[26] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. LLAMA: A Cache/Storage Subsystem for Modern Hardware. *Proceedings of the VLDB Endowment* 6, 10 (2013), 877–888.

[27] Ming Li, Wendi Cheng, Jiahe Wei, Xueqiang Shan, Weikai Liu, Xiaonan Zhao, and Xiao Zhang. 2025. BVLSM: Write-Efficient LSM-Tree Storage via WAL-Time Key-Value Separation. arXiv:2506.04678.

[28] Yongkun Li, Zhen Liu, Patrick P. C. Lee, Jiayu Wu, Yinlong Xu, Yi Wu, Liu Tang, Qi Liu, and Qiu Cui. 2021. Differentiated Key-Value Storage Management for Balanced I/O Performance. In *USENIX ATC*. USENIX Association, Virtual, 673–687.

[29] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *FAST*. USENIX Association, Santa Clara, CA, 133–148.

[30] Meta. 2021. BlobDB: RocksDB's Key-Value Separation for Large Values. Retrieved July 27, 2026 from https://github.com/facebook/rocksdb/wiki/BlobDB

[31] Meta. 2021. Integrated BlobDB. Retrieved July 27, 2026 from https://rocksdb.org/blog/2021/05/26/integrated-blob-db.html

[32] Meta. 2024. RocksDB: A Persistent Key-Value Store for Flash and RAM Storage. Retrieved July 27, 2026 from https://github.com/facebook/rocksdb

[33] Meta. 2024. RocksDB: Setup Options and Basic Tuning. Retrieved July 27, 2026 from https://github.com/facebook/rocksdb/wiki/Setup-Options-and-Basic-Tuning

[34] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems* 17, 1 (1992), 94–162.

[35] Mysten Labs. 2025. TideHunter: Source Code Repository. Retrieved July 27, 2026 from https://github.com/MystenLabs/tidehunter

[36] Michael A. Olson, Keith Bostic, and Margo I. Seltzer. 1999. Berkeley DB. In *USENIX ATC*. USENIX Association, Monterey, CA, 183–191.

[37] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385.

[38] OpenMetal. 2024. OpenMetal: On-Demand Private Clouds. Retrieved July 27, 2026 from https://openmetal.io

[39] Laurel Orr, Atindriyo Sanyal, Xiao Ling, Karan Goel, and Megan Leszczynski. 2021. Managing ML Pipelines: Feature Stores and the Coming Wave of Embedding Ecosystems. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3178–3181.

[40] Anastasios Papagiannis, Giorgos Saloustros, Giorgos Xanthakis, Giorgos Kalaentzis, Pilar Gonzalez-Ferez, and Angelos Bilas. 2021. Kreon: An Efficient Memory-Mapped Key-Value Store for Flash Storage. *ACM Transactions on Storage* 17, 1 (2021), 7:1–7:32.

[41] Chanyoung Park, Jungho Lee, Chun-Yi Liu, Kyungtae Kang, Mahmut Taylan Kandemir, and Wonil Choi. 2025. AnyKey: A Key-Value SSD for All Workload Types. In *ASPLOS*. ACM, Rotterdam, Netherlands, 47–63. https://doi.org/10.1145/3669940.3707279

[42] PingCAP. 2019. Titan: A RocksDB Plugin for Key-Value Separation. Retrieved July 27, 2026 from https://github.com/tikv/titan

[43] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *SOSP*. ACM, Shanghai, China, 497–514.

[44] Justin Sheehy and David Smith. 2010. *Bitcask: A Log-Structured Hash Table for Fast Key/Value Data*. Technical Report. Basho Technologies.

[45] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-Memory Databases. In *SOSP*. ACM, Farmington, PA, 18–32.

[46] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *SIGMOD*. ACM, Chicago, IL, 1041–1052.

[47] Maofan Yin, Hongbo Zhang, Robbert van Renesse, and Emin Gün Sirer. 2020. CedrusDB: Persistent Key-Value Store with Memory-Mapped Lazy-Trie. arXiv:2005.13762.

[48] Geoffrey X. Yu, Markos Markakis, Andreas Kipf, Per-Åke Larson, Umar Farooq Minhas, and Tim Kraska. 2022. TreeLine: An Update-In-Place Key-Value Store for Modern Storage. *Proceedings of the VLDB Endowment* 16, 1 (2022), 99–112.

[49] Benjamin Zhu, Kai Li, and R. Hugo Patterson. 2008. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *FAST*. USENIX Association, San Jose, CA, 269–282.