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) epoch-based pruning that reclaims space without blocking writes. On a 1 TB dataset with 1 KB values, Tidehunter achieves 830K writes per second, that is 8.4× higher than RocksDB and 2.9× higher than BlobDB, while improving point queries by 1.7× and existence checks by 15.6×. 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.

1 INTRODUCTION

Log-Structured Merge-Trees (LSM-trees) are the dominant design for persistent key-value stores. An LSM-tree buffers writes in memory and flushes them to disk as sorted files. To bound the number of
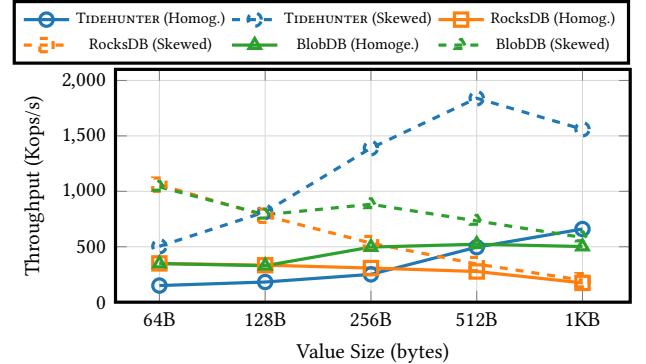
Figure 1: Mixed workload throughput (50% reads/writes) vs. value size, with skewed (Zipf $\theta = 2$) and homogeneous (Zipf $\theta = 0$) access patterns.

files and maintain lookup performance, the storage engine periodically merges these files in a process called compaction. Compaction rewrites data to maintain sorted order. For small values, the overhead is acceptable. For values of several kilobytes, it is not. A common measure of this inefficiency is write amplification, i.e., bytes written to disk divided by bytes received from the application. Write amplification reaches 10× to 30× [17] 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.

LSM-trees target hard disk drives (HDDs), where sequential I/O is 100× faster than random, hence trading write amplification for sequential access was the recommended approach. 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. To make matters worse, SSD cells also wear out after a finite number of writes, making high write amplification the culprit of frequent device failures.

A seminal work to address this issue is WiscKey [26]. It 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. Additionally, the value log of WiscKey needs garbage collection, which stalls I/O.

We take a different path. In Tidehunter, the Write-Ahead Log (WAL), traditionally a temporary buffer for crash recovery, becomes the permanent store for values. We replace the key LSM-tree with sharded index tables that flush lazily and answer existence queries from memory. Our design targets applications with large values and uniformly distributed keys. These are common requirements

of modern applications. Content-addressable storage systems like Git and IPFS key objects by cryptographic hashes of their contents [4]. Deduplication systems for backup and archival storage partition data into chunks and index them by hash, with chunk sizes typically ranging from 4 KB to 64 KB [19, 39]. Object stores and blob storage systems often use UUIDs or content hashes as keys; Facebook's Haystack, for example, stores billions of photos keyed by unique identifiers with a size of around 8-64 KB [3]. Machine learning feature stores index embeddings and feature vectors by entity ID or feature hash, with vectors often reaching several kilobytes [15, 32, 33]. The shift toward immutable, content-addressed, and UUID-keyed data spans cloud infrastructure, data pipelines, and application backends. 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. A validator participates in distributed consensus: it receives transactions, orders them into blocks, executes them, and stores results for queries. We specifically chose Sui [6] to integrate TIDEHUNTER. It is a high-throughput blockchain whose validators handle thousands of transactions per second and answer queries about history, execution results, and object state concurrently. Validators write continuously as blocks arrive but must read with low latency, since execution fetches objects, clients check status, and peers request proofs. They also prune aggressively, keeping only recent epochs; this places significant stress on the underlying RocksDB store, where ill-timed compactions can cause performance to plummet.

TIDEHUNTER is built for large values and uniform keys. Figure 1 shows the performance benefit of TIDEHUNTER as the value size grows, both for skewed and homogeneous access patterns. Three key insights guided our design towards these results:

(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 epoch semantics for garbage collection.** Many workloads, including blockchains, organize data into epochs or time ranges. TIDEHUNTER reclaims space by dropping entire WAL segments when their epoch expires, avoiding the I/O cost of per-record garbage collection.

We evaluate TIDEHUNTER against RocksDB and BlobDB (the production implementation of WiscKey) on 1 TB datasets. At 1 KB values, TIDEHUNTER sustains 830K writes/sec (8.4× RocksDB, 2.9× BlobDB), improves point queries by 1.7×, and improves existence checks by 15.6× (resolved from the index, no value fetch). At 64 bytes, LSM-trees win by 2–2.5×; the crossover is near 128 bytes.

To validate that synthetic benchmarks translate to real systems, we integrated TIDEHUNTER into Sui and ran validators under sustained transaction load. With RocksDB, validators show performance degradation at around 6,000 transactions per second as disk utilization rises and latency climbs. Under the same conditions,

TIDEHUNTER maintains stable throughput and lower latency, with reduced disk pressure and more consistent CPU utilization. Side-by-side runs show clear differences in disk read and write throughput, sustained transactions per second, and settlement finality latency. TIDEHUNTER avoids the I/O saturation that triggers performance collapse in RocksDB configurations at comparable load.

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, showing 8.4× write throughput over RocksDB and stable performance where RocksDB-backed systems collapse.

2 BACKGROUND & DESIGN PRINCIPLES

We begin by reviewing LSM-trees and their compaction costs, which motivate TIDEHUNTER's design. We then present the high-level principles that guide TIDEHUNTER's architecture.

2.1 Log-Structured Merge-Trees

LSM-trees [30] emerged as the dominant architecture for write-intensive key-value stores. The design reflects a fundamental asymmetry of hard disk drives: sequential I/O is roughly 100× faster than random I/O [20, 37]. Rather than update records in place, which requires random seeks, LSM-trees buffer writes in memory and flush them sequentially. This batching converts random writes into sequential at the cost of extra bookkeeping.

An LSM-tree organizes data into levels. Incoming writes first land in *memtable*, an in-memory sorted structure such as a balanced tree or skiplist. When the memtable reaches a size threshold, the engine flushes it to disk as an immutable Sorted String Table (SST) file. These flushed files form Level 0 (L0). Unlike deeper levels, L0 files may have overlapping key ranges because each flush produces an independent file. As L0 accumulates files, read performance degrades: a lookup checks the memtable, then each L0 file from newest to oldest, because any of them might contain the key. To bound this, the engine periodically runs *compaction*, which merges L0 files into Level 1 (L1). L1 and deeper levels maintain a key invariant: within a level, files have non-overlapping key ranges. A lookup at L1 or beyond touches aa single file per level. Compaction continues down the tree: when L1 grows too large, files merge into L2, and so on.

RocksDB [17], a widely deployed LSM-tree implementation, uses this leveled compaction strategy by default. Each level is roughly 10× larger than the previous, so a dataset of size N requires $O(\log N)$ levels. Compaction is crucial for read performance: without it, every lookup would scan the entire L0, making reads linear in the number of flushes.

2.2 Compaction Costs

Compaction maintains read performance but introduces three forms of overhead that dominate under write-heavy workloads.
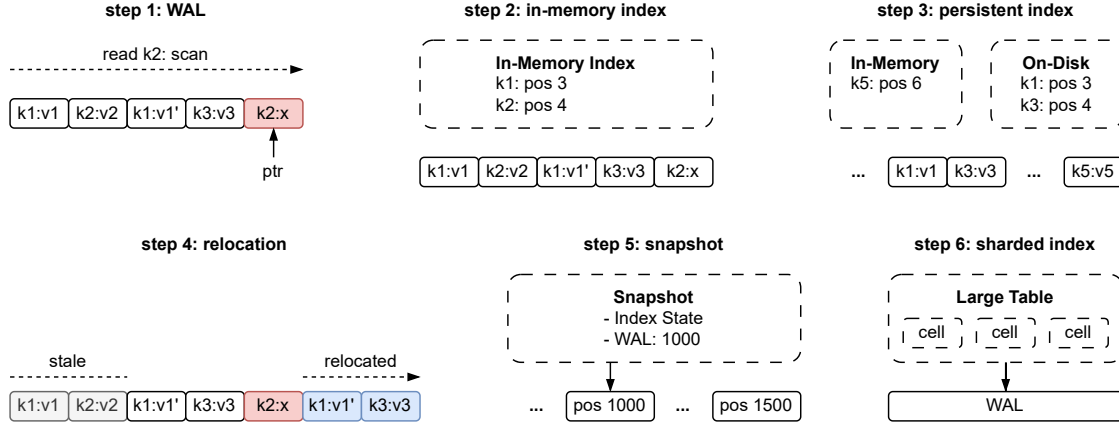
Figure 2: Design evolution of TIDEHUNTER. Each step addresses a limitation of the previous design: (1) append-only log provides fast writes but slow reads; (2) in-memory index enables fast lookups; (3) persisted index supports datasets larger than memory; (4) relocation reclaims space without blocking writes; (5) snapshots enable fast recovery; (6) sharding enables concurrent operations.

Write amplification. Each key-value pair may be rewritten many times as it moves through levels. When compaction merges files from Li into L$i + 1$, every record in those files is read and written again, even if it has not changed. With a size ratio of 10× between levels and $O(\log N)$ levels, write amplification—bytes written to disk divided by bytes received from the application—can reach 10× to 30× under random workloads [17]. In Sui's private testnet, validators ingesting 40 MB/s of application data generated 500 MB/s of disk writes, an 8× amplification factor. High write amplification consumes I/O bandwidth, accelerates SSD wear [22], and limits throughput scaling.

Read amplification. Although compaction reduces the number of files a lookup must check, reads still traverse multiple levels. A point query may touch the memtable, one or more L0 files, and one file per deeper level. Bloom filters mitigate this cost for negative lookups [17], but positive lookups on cold data incur one disk read per level. For a 1 TB dataset with 10× level ratios, this means five or six levels and potentially five or six disk reads per lookup.

Write stalls. Compaction competes with foreground writes for I/O bandwidth and CPU cycles. To prevent L0 from growing without bound, RocksDB throttles or blocks writes when compaction falls behind. These *write stalls* cause latency spikes and throughput drops [2, 17]. In extreme cases, writes block entirely until compaction clears enough space. The result is unpredictable performance: the same workload may sustain high throughput one minute and stall the next. These costs compound as the dataset grows and values become larger. In Sui's testnet, disk I/O utilization approached 80%, leaving no headroom to scale throughput. The bottleneck was not the application—it was the storage engine rewriting data that had not changed. TIDEHUNTER addresses this by eliminating value compaction entirely: values land in the WAL once and never move again.

2.3 TIDEHUNTER Design Principles

The limitations of LSM-Tree-based designs specifically for applications with large values and uniform keys has guided our design for

TIDEHUNTER. Figure 2 presents our journey towards TIDEHUNTER by starting from a minimal storage system and incrementally addressing its limitations. Each step introduces a component that appears in the final architecture (Figure 2). 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.

Step 1: Append-Only Log. The simplest design stores all data in a single append-only log. Inserts and updates append the key-value pair to the end of the log. Deletes append a tombstone marker. Reads perform a linear scan from the end of the log, returning the first matching key encountered. This design achieves $O(1)$ writes but $O(n)$ reads. This is acceptable for write-heavy workloads with few reads, but impractical otherwise. We need an index.

Step 2: In-Memory Index. We add an in-memory index that maps keys to their positions in the log. Writes still append to the log, but now also update the index. Reads consult the index to find the log position, then fetch the value directly—$O(\log n)$ or $O(1)$ depending on the index structure. However, the index grows with the number of unique keys. For datasets larger than memory, the index itself becomes the bottleneck. We need to persist it.

Step 3: Persisted Index. We periodically flush the in-memory index to disk. The flushed index is a sorted file that can be searched efficiently. Reads first check the in-memory index for recent writes, then fall back to the persisted index on disk. Now the system can handle datasets larger than the RAM. But the WAL grows unbounded: deleted keys and overwritten values consume space forever. We need to reclaim it.

Step 4: Relocation. We introduce a relocation process that moves live entries forward in the log and discards obsolete ones. The relocator scans from the oldest entries, checks whether each key is still live (not deleted or overwritten), and re-appends live entries to the log tail. Once all live entries in a log segment have been relocated, the segment can be deleted. Crucially, relocation runs in the background and never blocks writes. Unlike LSM-tree compaction,
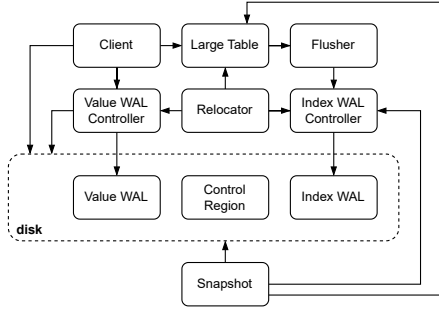
Figure 3: High-level architecture of TIDEHUNTER.

which must complete before the system can proceed, TIDEHUNTER's relocator simply appends to the same log that serves normal writes. However, recovery currently requires replaying the entire WAL to reconstruct the index, which is impractically slow for large datasets.

Step 5: Snapshots. We add periodic snapshots that record the index state and a WAL position on disk. Recovery loads the snapshot and replays only the WAL suffix after the snapshot position. Snapshots are small because they store only metadata (index positions and WAL offsets), not the actual index data. They can be taken frequently without significant overhead. Finally, a single index becomes a contention point under concurrent writes: locking it even for a brief time for updates can severely hinder performance.

Step 6: Sharded Index. We partition the key space into independent cells, each with its own in-memory index, persisted index, and locks. Operations on different shards proceed in parallel without synchronization. This is TIDEHUNTER in a nutshell. The WAL remains shared (writes are serialized only at allocation), but index operations parallelize across shards. Section 3 describes each component and the challenges that this design overcomes in detail.

3 THE TIDEHUNTER ARCHITECTURE

Figure 3 illustrates the architecture. The system consists of several interacting components. This section describes the interaction between these components to serve reads and writes as well as ensuring efficient crash-recovery.

3.1 Write Flow

Figure 4 details the write path by expanding the Value WAL Controller of Figure 3. The write path is designed to minimize latency while providing clear durability guarantees. It contains a synchronous path that completes fast (solid arrows on Figure 4), and asynchronous background operations (dotted arrows on Figure 4).

The Client flow. The client starts by sending a write operation (insert or delete) to the synchronous controller, which allocates a position in the Value WAL, that is, a contiguous region where the entry will be written, to ensure concurrent writers do not overwrite each other (❶). Its role is to monitor which WAL positions have been fully processed (i.e., both written to the WAL and indexed in the Large Table). It then memory-map writes the entry to the Value WAL, persisting it to disk (❷). The client then updates the Large Table, the in-memory index that maps keys to their WAL
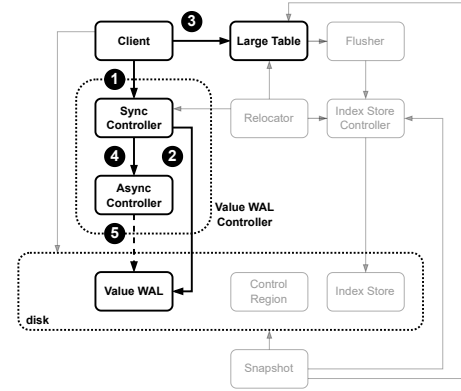

Figure 4: Write path separating fast synchronous operations (solid) from background durability (dotted).

positions, so that subsequent reads can locate the value directly without scanning the entire WAL (❸). This allows multiple clients to start the write flow concurrently. The synchronous controller allocates WAL positions through a fast atomic counter and after that, the clients run the rest of the write flow in parallel.

When the update completes, the asynchronous controller is notified that this position is now fully processed, meaning the entry is both durable in the WAL and indexed in the Large Table (❹). It updates the highest WAL offset for which all preceding entries have been processed. This position, along with a snapshot of the Large Table, is periodically persisted to the Control Region to facilitate efficient crash recovery (see Section 3.4).

The Value WAL is organized into memory-mapped regions called maps, each covering a contiguous range of positions. Multiple entries are written to the same map before it is finalized. When all positions in a map have been processed and the next map has begun receiving writes, the asynchronous controller asynchronously finalizes the map. The asynchronous controller serves three purposes: it manages the lifecycle of these memory maps, handles garbage collection of old WAL files, and calls fsync to finalize maps. For map lifecycle, it maintains a buffer of pre-allocated maps so that writes never block waiting for map creation. When a map is finalized, it prepares the next map in the buffer; it then asynchronous calls fsync on finalized maps, ensuring data is persisted to stable storage (❺). By isolating this blocking operation in background threads, the write path remains fast and non-blocking. For garbage collection, when relocation advances the minimum required WAL position, the asynchronous controller deletes obsolete WAL files that are no longer needed for recovery (see Section 4.4).

Durability Guarantees. This design cleanly separates the synchronous write path: allocation, memory-map write, and index update, from the asynchronous durability and cleanup operations. Writes complete without waiting for fsync, while background threads ensure data eventually reaches stable storage and stale files are reclaimed. TIDEHUNTER provides durability against application crashes as soon as step ❸ completes, when data is written to the memory-mapped region. Even if the application crashes before the Large Table is updated, the kernel will eventually flush the page
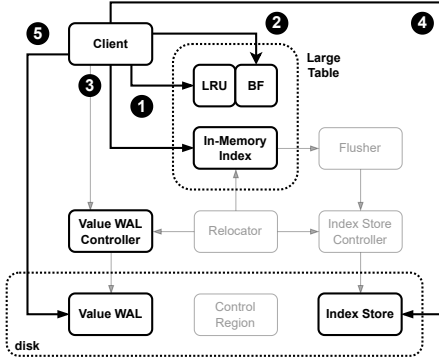
Figure 5: Read path showing the tiered lookup strategy. The snapshot component is omitted for clarity.

cache to disk. Upon recovery, TIDEHUNTER replays the WAL from the last checkpointed position, reconstructing any index entries that were not yet reflected in the Large Table. For durability against kernel crashes, TIDEHUNTER relies on explicit fsync operations performed asynchronously by the background asynchronous controller. Applications requiring stronger durability guarantees can invoke explicit flush operations. Having the synchronous controller separated from the asynchronous controller allows for a multi-tier performance/durability trade-off. The former finishes fast and provides persistence at the OS page cache, while the latter batches fsync calls and handles cleanup in the background.

Deletes. Deleting a key is analogous to writing a special tombstone value. If concurrent operations occur (e.g., a delete and an insert for the same key), the operation with the higher WAL position wins. This ensures consistency regardless of the order in which operations are applied to the Large Table. Tombstones serve three purposes: they ensure deletes survive crash recovery (replaying the tombstone removes the key), they allow correct conflict resolution with concurrent writes, and they enable the relocator to skip deleted entries when compacting the Value WAL (Section 4.4).

Atomic batch writes. TIDEHUNTER supports atomic batch writes, allowing multiple inserts and deletes to be applied as a single unit. A batch is constructed by accumulating updates in memory, then committed in two phases. First, a batch-start marker is written to the Value WAL, followed by all entries in the batch. The Value WAL Controller uses a single allocation for the entire batch, ensuring they are treated as a unit for position tracking. Then, each entry is applied to the Large Table in sequence. If a crash occurs after the WAL write but before all Large Table updates complete, recovery replays the batch: the batch-start marker indicates how many entries follow, ensuring partial batches are reconstructed. This provides atomicity, either all or no update in the batch is visible after recovery.

3.2 Read Flow

Figure 5 shows the read path in TIDEHUNTER by expanding the Large Table of Figure 3. When a client issues a read operation, TIDEHUNTER follows a tiered lookup strategy.

It first consults an in-memory LRU cache that stores recently accessed values (❶). If the key is found, the value is returned immediately. This is the fastest path, requiring no index lookup or WAL access. If the LRU cache misses, the system checks a bloom filter (❷). If the bloom filter indicates the key is definitely not present, the lookup returns immediately, avoiding unnecessary index access. This is fundamentally different from RocksDB-style LSMs, where queries traverse multiple levels of SSTables regardless of whether the key exists. If the bloom filter does not rule out the key, the system consults the Large Table (❸). If the relevant index is resident in memory, the lookup completes immediately; otherwise, the system reads the index from the Index Store on disk (❹). If the key is found, the index provides a WAL position, and the system reads the value directly from the Value WAL at that position (❺). The value is then added to the LRU cache to accelerate future accesses. In practice, frequently accessed data remains in the operating system's page cache, making most reads memory-speed operations.

When a cell's index resides on disk rather than in memory, the system must typically load the entire index before performing a lookup. However, for uniformly distributed keys, TIDEHUNTER can locate entries directly on disk without loading the full index. For large cells, this is wasteful when only a single key is needed. TIDEHUNTER exploits a key property of its workload: keys are either cryptographic hashes or prefixed by such hashes, and thus uniformly distributed across the keyspace. This allows the system to estimate where a key would appear in a sorted, serialized index. The algorithm treats the key as an integer and computes its fractional position within the full key range, for example, a key starting with byte 0x80 lies roughly at the midpoint. Multiplying this fraction by the index file size yields an estimated byte offset.

The system then reads a small window around this offset. If the target key falls within the range covered by the window, binary search locates the entry. Otherwise, the window is shifted toward the beginning or end of the file based on key comparison, and the process repeats. For uniformly distributed keys, this typically converges within 1-3 iterations, reading only a few kilobytes rather than potentially megabytes of index data. We call this approach the *optimistic index* and detail its statistical foundation in Section 4.2. This optimization is particularly valuable during recovery and cold-start scenarios, where most cells are unloaded. It allows TIDEHUNTER to serve read requests immediately after opening the database, without waiting for indices to be loaded into memory (see Section 3.4).

3.3 State Snapshots

TIDEHUNTER periodically captures snapshots in the background to enable efficient crash recovery. A snapshot records the state of the system at a point in time, allowing recovery to start from that point rather than replaying the entire Value WAL from the beginning. For each cell in the Large Table, a snapshot stores the Index Store position where the cell's most recently flushed index resides (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 the starting point for Value WAL replay during recovery. The snapshot engine operates as a background thread that periodically

flushes cells that have accumulated significant updates and writes this metadata atomically to the Control Region, a dedicated region on disk (Figure 3). More frequent snapshots reduce recovery time by minimizing the Value WAL segment that must be replayed, but increase write I/O during normal operation; TIDEHUNTER balances this trade-off by flushing only cells that exceed a configurable dirty threshold. Unlike LSM-trees based systems [28] that snapshot entire data structures, TIDEHUNTER snapshots contain only positions rather than actual index data. As a result, they are small and can be rewritten frequently without significant overhead.

3.4 Crash Recovery

When TIDEHUNTER starts after a crash or normal shutdown, it performs a recovery sequence. The system first reads the Control Region from disk. This contains the most recent snapshot: for each cell, the Index Store position where its flushed index resides, and a global replay-from position marking where Value WAL replay must begin. The system then opens both the Value WAL and the Index Store. These append-only logs survived the crash; their contents remain intact. The Large Table is initialized from the snapshot metadata. Each cell starts in an unloaded state, but the cell knows where its serialized index resides in the Index Store. This allows recovery to proceed without loading all indices upfront as they are loaded on demand during normal operation. Starting from the replay-from position recorded in the snapshot, the system iterates through the Value WAL and replays each entry into the Large Table, reconstructing any updates that occurred after the snapshot was taken. Once replay completes, the system starts the background components: the Value WAL Controller, the Index Store Controller, the flusher threads, the relocator, and the snapshot thread. The database is now ready to serve client requests.

4 RESOURCE MANAGEMENT

This section describes how TIDEHUNTER manages resources to scale beyond available memory. We examine the Large Table's organization, which allows the index to partially reside on disk; index flushing, which persists updates to relieve memory pressure; and relocation, which reclaims disk space by pruning obsolete entries without blocking writes.

4.1 The Large Table

The Large Table mainly serves as TIDEHUNTER's in-memory index, mapping keys to their corresponding positions in the Value WAL. Its design enables memory-efficient operation by allowing portions of the index to reside on disk while supporting concurrent access through fine-grained locking.

Partitioning and concurrency. Supporting datasets larger than memory while maintaining high concurrency presents two challenges: the index cannot fit entirely in RAM, and a single lock would serialize all operations. TIDEHUNTER addresses both by partitioning keys into cells, each representing a contiguous range of keys. This partitioning enables two key properties: cells can be individually loaded or evicted to manage memory, and cells can be locked independently to maximize concurrency. Applications define one or more key spaces at database creation, each configured with a

distribution type. For uniformly distributed keys (such as cryptographic hashes), cells are pre-allocated as a fixed-size array. For keys with common prefixes, cells are stored in a B-tree that grows dynamically. This latter organization suits composite keys such as (tenant_id, record_id) or (namespace, key), enabling efficient range scans within a prefix while accommodating new prefixes 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 governing memory residency. A cell starts *Empty* and transitions to *Loaded* when it receives data that is fully resident in memory. When evicted under memory pressure, it becomes *Unloaded*, with its index existing only on disk. *DirtyLoaded* indicates an in-memory index with unpersisted changes, while *DirtyUnloaded* buffers recent writes while the bulk of the index remains on disk. When a write arrives for an Unloaded cell, the system transitions to DirtyUnloaded and buffers only the new entry. Reads check this in-memory buffer first; if the key is not found, the system performs a point lookup into the on-disk index without loading the entire structure (Section 3.2). This trades slightly higher per-lookup cost for substantially lower memory consumption.

4.2 Optimistic Index Lookup

When a cell's index resides on disk, lookups typically require loading the index into memory or using a multi-level directory structure requiring multiple I/O operations. TIDEHUNTER avoids both by exploiting a statistical property of uniform key distributions: the position of a key 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 format is deliberately simple: a sorted array of fixed-size entries, each containing a 32-byte key and an 8-byte WAL position (40 bytes total). No headers, directories, or multi-level structures are needed. This simplicity enables direct position estimation: given a key k and an index containing N entries, the system computes the fractional position $p = k/2^{256}$ and estimates the byte offset as $p \times 40N$.

To perform a lookup, the system reads a window of W entries (800 by default, approximately 32 KB) centered at the estimated offset. If the target key falls within the key range covered by the window, binary search locates the entry. If not, the window shifts toward the beginning or end of the file based on key comparison, and the process repeats. For uniformly distributed keys, this typically converges in one to three iterations. The approach works because modern SSDs exhibit a batch read property: reading 32 KB incurs essentially the same latency as reading a single byte, due

to internal page sizes and command overhead. A window of 800 entries provides greater than 99% probability of containing the target key for indices with 10K–100K entries. The window size is configurable, trading off between hit rate (larger windows miss less often) and I/O volume (smaller windows read fewer bytes per iteration). We evaluate this trade-off in Section 6.

4.3 Index Flushing

When a cell accumulates too many dirty entries, the system must persist them to disk. A naive approach would block the cell during this flush, but this is unacceptable for write-heavy workloads where cells receive continuous updates. TIDEHUNTER flushes asynchronously: the flusher captures a snapshot of the dirty index and dispatches persistence to background threads, while the cell remains available for new writes.

The flusher performs one of two operations depending on the cell's state: for DirtyLoaded cells, it serializes the complete in-memory index; for DirtyUnloaded cells, it first loads the existing on-disk index, merges it with the dirty entries, and serializes the combined result. The flusher may also remove obsolete entries. Once the index is prepared, the flusher hands it to the Index Store Controller for persistence. The Index Store shares the same append-only implementation as the Value WAL (Section 3.1), differing only in the data it stores (serialized indices rather than key-value entries). It then notifies the Large Table with the new index position, so the cell can record where its on-disk index is stored for future reads. The cell performs an unmerge operation: it removes from its in-memory buffer all entries included in the flush, retaining only entries that arrived after the flush began. If no entries remain, the cell becomes to Unloaded; otherwise, it remains in DirtyUnloaded with the updated on-disk pointer.

A second challenge is serving reads during a flush. Because the Index Store is append-only, new indices are appended rather than overwriting existing ones. During a flush, the cell continues pointing to the previous index position, so concurrent reads safely access the old version. Only after the write completes does the flusher atomically update the cell's pointer. Readers and writers thus operate on disjoint regions, requiring no coordination beyond the final pointer update.

4.4 Relocation

As writes accumulate in the Value WAL, obsolete entries, overwritten values and deleted keys, consume space that can no longer be reclaimed by simply truncating the log. TIDEHUNTER reclaims this space through a process called relocation, which moves live entries forward in the WAL, allowing old segments to be deleted. Relocation operates by scanning the WAL, identifying entries that are still live, re-writing them to new positions at the tail of the WAL, updating the Large Table index to point to the new positions, and finally deleting the old WAL segments. The key challenge is ensuring correctness in the presence of concurrent writes: if a client updates a key while relocation is in progress, the relocated entry must not overwrite the newer value. TIDEHUNTER solves this with a compare-and-set mechanism. When relocation reads an entry at position P and prepares to relocate it to position P', it captures the last processed value L—the highest contiguous WAL position

where all write operations have been written to the WAL and their corresponding index updates have been applied to the Large Table. 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 allows relocation to run concurrently with normal writes without locks or coordination. 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, ensuring files are not deleted prematurely.

TIDEHUNTER supports two relocation strategies: *WAL-based relocation* and *index-based relocation*. WAL-based relocation scans the Value WAL sequentially from the oldest entry. For each entry, it checks whether the key still exists in the index and points to this WAL position—if so, the entry is live and must be relocated. This approach is simple and processes entries in WAL order, but must read each entry from disk. Index-based relocation iterates through cells in the Large Table. For each cell, it loads the index, identifies entries whose WAL positions fall below the target cutoff (a configurable position that determines how much of the WAL to reclaim), reads their values from the WAL, and decides which to relocate. This approach can be more efficient when only a subset of cells contain old entries. Both strategies support an optional relocation filter, an application-provided callback that decides for each entry whether to keep it, remove it, or stop relocation. This enables application-specific compaction logic, such as retaining only the latest version of each key or pruning entries based on business rules.

A key architectural difference between TIDEHUNTER and LSM-tree databases like RocksDB is the role of compaction. In LSM-trees, compaction is essential for read performance: without it, reads must search through multiple levels of SSTables, and read amplification grows with the number of levels. Compaction merges levels, reducing the search space and keeping reads fast. In TIDEHUNTER, read performance is independent of WAL fragmentation. The Large Table index provides direct access to any value's WAL position—whether that position is at the beginning or end of the WAL, the read cost is the same (one index lookup plus one WAL read). Fragmentation does not degrade read latency. This means relocation in TIDEHUNTER serves a single purpose: reclaiming disk space. Unlike LSM-tree compaction, which repeatedly rewrites data across levels, TIDEHUNTER writes each value to the WAL only once during normal operation. Additional writes occur only when relocation reclaims space, and even then, only live entries are rewritten. Because relocation is optional and can be deferred when disk space is abundant, applications can maintain near-1× write amplification.

Unlike LSM-tree compaction, which trades write amplification for read performance, TIDEHUNTER's relocation trades write amplification purely for disk space. Applications with ample storage can defer or skip relocation entirely without impacting read latency.

5 IMPLEMENTATION

We provide a *production-ready* implementation of TIDEHUNTER in Rust, currently being deployed in the Sui blockchain to replace

RocksDB, comprising approximately 14,000 lines of code (excluding tests). The implementation leverages several key libraries that align with the system's design goals: *memmap2* for memory-mapped I/O, enabling direct writes to WAL fragments without explicit system calls; *parking_lot* for high-performance mutexes with support for Arc-based guards, used in the guard-based position tracking mechanism; *arc-swap* for atomic swapping of Arc pointers, allowing lock-free reads of shared data structures such as the WAL's memory map registry; *crc32fast* for efficient CRC32 checksums on WAL entries; *bloom* for per-cell Bloom filters that accelerate negative lookups; *lru* for the value cache; and *rayon* for parallel initialization of the Large Table during recovery. Unsafe code is limited to memory-mapped I/O operations and carefully encapsulated within the WAL module. The codebase includes approximately 160 unit tests, stress tests with shadow-state verification, and a failpoint infrastructure for deterministic concurrency testing.

TIDEHUNTER employs a fixed set of background threads to handle asynchronous operations. Each WAL Controller (Value and Index) runs three dedicated threads: a *tracker* that monitors position completion via the guard-based mechanism, a *mapper* that manages memory-mapped region lifecycle and garbage collection, and a *syncer* that issues fsync calls on finalized fragments. A configurable pool of *flusher* threads handles index persistence, allowing multiple cells to be flushed concurrently. A single *relocator* thread performs background compaction, moving live entries forward in the Value WAL to reclaim space. Finally, a *snapshot* thread periodically captures consistent checkpoints of the Large Table for crash recovery. In total, a typical deployment runs 8 background threads plus the configured number of flushers. The design deliberately avoids per-client or per-request thread spawning to minimize scheduling overhead and maintain predictable resource usage.

6 EXPERIMENTAL EVALUATION

We answer the following questions about TIDEHUNTER:

(1) How does the read-write ratio influence the performance of TIDEHUNTER?
(2) How does the type of read operation (get, exists, scan) influence the performance of TIDEHUNTER?
(3) How does workload skew influence the performance of TIDEHUNTER?
(4) How does value size influence the performance of TIDEHUNTER?
(5) How does relocation influence the performance of TIDEHUNTER?
(6) How does our optimistic index perform against the naive index?

6.1 Experimental Setup

We run our experiments on a Medium v4 OpenMetal instance [31] 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 the following state-of-the-art key-value stores:

- **RocksDB** [28]: An LSM-tree-based embedded key-value store widely used in production systems.

- **BlobDB** [27]: RocksDB's integrated blob storage extension inspired by WiscKey [26], which separates large values from the LSM-tree to reduce write amplification.

Methodology. We evaluate TIDEHUNTER against the baselines using a custom benchmark consisting of two phases: a *fill phase* and a *measurement phase*. In the fill phase, we populate the database with random keys and values. We use a fixed key size of 32 bytes, while the value size is a configurable parameter. We insert entries until the database reaches 1 TB of data; specifically, we divide 1 TB by the entry size (key size plus value size) to determine the number of insertions. After the fill phase, we allow a 10-minute cooldown period for the system to stabilize (e.g., for relocation to settle). In the measurement phase, we perform operations for a fixed duration of 10 minutes. The following workload parameters are configurable:

- **Read-write ratio**: We evaluate three configurations—100% writes, 50% reads with 50% writes, and 100% reads.
- **Read operation type**: We consider three read operations: *get* (returns the value for a key if it exists), *exists* (returns whether a key exists), and *reverse iterator* (returns the key-value pair with the largest key smaller than the query key, if one exists).
- **Skew**: A homogeneous workload selects keys uniformly at random, while a skewed workload favors more recently inserted keys. We implement skew using a Zipfian distribution with parameter $\theta = 0$ (homogeneous) or $\theta = 2$ (skewed). We use "homogeneous" rather than "uniform" for access patterns to avoid confusion with uniformly distributed keys.
- **Value size**: We evaluate both large values (1 KB) and small values (64 bytes and 128 bytes).
- **Relocation**: Relocation can be enabled or disabled to measure its impact on performance. Unless stated otherwise, all benchmark results were obtained with relocation enabled.
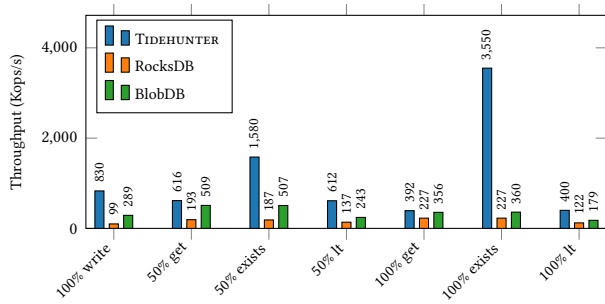
6.2 Benchmark Results

We present benchmark results for TIDEHUNTER, RocksDB, and BlobDB across various workload configurations.
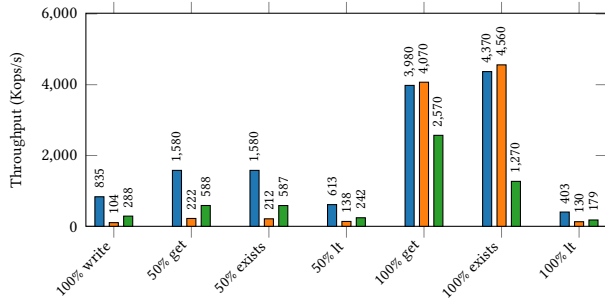
6.2.1 Large Values. The results for large values are in Figure 6.

Write Performance. Under write-only workloads (Figure 6a, leftmost bars), TIDEHUNTER achieves 830K ops/sec, outperforming RocksDB by 8.4× and BlobDB by 2.9×. This substantial 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 read workloads, TIDEHUNTER maintains a consistent advantage across all operation types. On get operations (Figure 6a), TIDEHUNTER achieves 392K ops/s compared to RocksDB's 227K ops/s, a 1.7× improvement. The exists operation (Figure 6a) reveals an even more pronounced advantage: TIDEHUNTER reaches 3.55M ops/s, outperforming RocksDB by 15.6×. This dramatic difference occurs because TIDEHUNTER can resolve existence queries directly from the index without fetching the actual value, while LSM-trees must traverse multiple levels regardless of whether the value is needed.

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

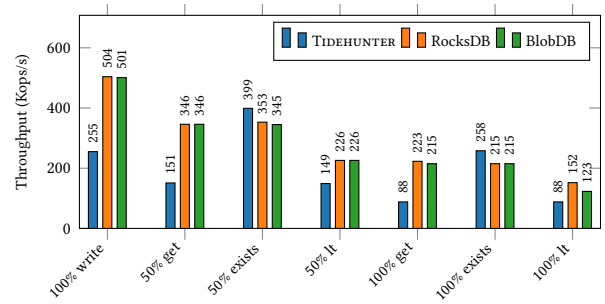(b) Skewed Distribution ($\theta = 2$)

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

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

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

Figure 7: Throughput comparison for small values (64 bytes). The 50% read cases include 50% writes. Top: homogeneous access distribution ($\theta = 0$). Bottom: skewed distribution ($\theta = 2$).

For reverse iterator operations (Figure 6a), TIDEHUNTER maintains a 3.3× advantage over RocksDB (400K vs 122K ops/s). However, all systems show reduced throughput compared to point queries, as range operations require examining multiple entries.
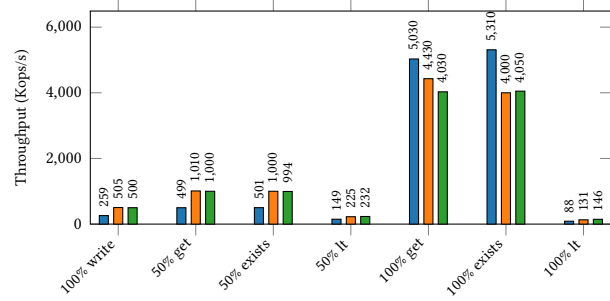
Mixed Workloads. At a 50/50 read-write ratio, TIDEHUNTER continues to lead with 616K ops/s for get operations, compared to RocksDB's 193K ops/s. The performance gap narrows relative to write-only workloads but remains substantial (3.2×), demonstrating that TIDEHUNTER's design also benefits mixed read-write 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, RocksDB achieves 4.07M ops/s, approaching TIDEHUNTER's 3.98M ops/s. This convergence occurs because 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. However, TIDEHUNTER maintains its advantage for write-heavy and mixed workloads even under skew, as its write path remains fundamentally more efficient.

Summary. For large values, TIDEHUNTER consistently outperforms LSM-tree-based alternatives. The advantage is most pronounced for write-heavy workloads (up to 8.4×) and existence checks (up to 15.6×), and remains significant for read-heavy homogeneous workloads (1.7×). These results validate TIDEHUNTER's design goal of minimizing data movement.

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 achieve approximately 500K ops/s, outperforming TIDEHUNTER's 255K ops/s by nearly 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 small, collocated entries.

Read Performance. The gap widens for read operations under homogeneous access patterns. For get operations at 100% reads (Figure 7a, rightmost bars), RocksDB achieves 223K ops/s compared to TIDEHUNTER's 88K ops/s—a 2.5× advantage. This difference reflects LSM-trees' superior cache efficiency: sorted runs pack many small entries into each cached block, whereas TIDEHUNTER's position-based lookups may incur more cache misses when entries are scattered across the WAL. Notably, TIDEHUNTER's exists operation (Figure 7a) shows improved relative performance, achieving 258K ops/s at 100% reads compared to RocksDB's 215K ops/s. This demonstrates that TIDEHUNTER's 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
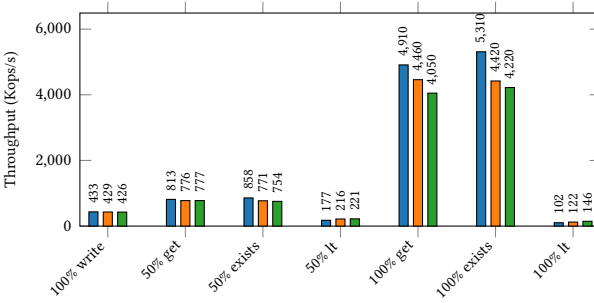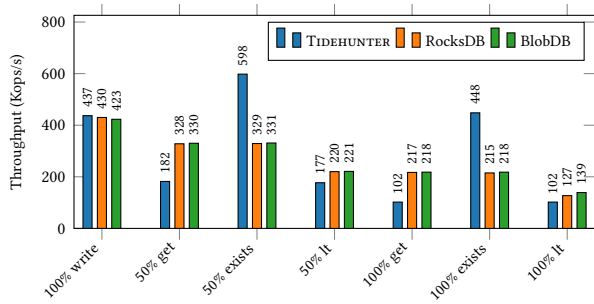
(a) Homogeneous Distribution ($\theta = 0$)

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

Figure 8: **Throughput comparison for small values (128 bytes). The 50% read cases include 50% writes. Top: homogeneous access distribution ($\theta = 0$). Bottom: skewed distribution ($\theta = 2$).**
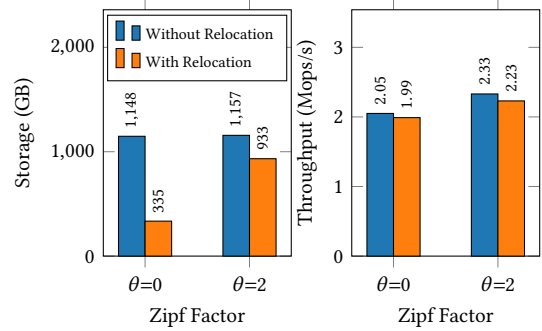

Figure 9: Effect of relocation on storage usage and throughput. Left: storage after mixed phase. Right: throughput during mixed phase. Relocation significantly reduces storage with minimal throughput impact.

Read Performance. For homogeneous read workloads, LSM-trees maintain an advantage for value-fetching operations. At 100% reads, RocksDB achieves 217K ops/s for get operations compared to TIDE-HUNTER's 102K ops/s (Figure 8a). However, TIDEHUNTER excels at existence checks: at a 50/50 read-write ratio, TIDEHUNTER achieves 598K ops/s for exists operations, outperforming RocksDB's 329K ops/s by 1.8× (Figure 8a). This demonstrates that TIDEHUNTER's optimized existence check provides substantial benefits.

For reverse iterator operations (Figure 8a), TIDEHUNTER leads under write-heavy workloads (439K vs 382K ops/s) but trails under read-heavy workloads (102K vs 127K ops/s), reflecting the mixed nature of this transitional value size.

Effect of Skew. Under skewed access patterns (Figure 8b), TIDE-HUNTER's in-memory caching provides a consistent advantage for read-heavy workloads. For get operations at 100% reads with skew, TIDEHUNTER achieves 4.91M ops/s versus RocksDB's 4.46M ops/s (10% improvement). The advantage is more pronounced for exists operations, where TIDEHUNTER reaches 5.31M ops/s compared to RocksDB's 4.42M ops/s (20% improvement).

Summary. At 128 bytes, TIDEHUNTER and LSM-tree systems show more balanced performance compared to smaller values. TIDE-HUNTER matches or exceeds LSM-trees for write-heavy workloads and existence checks, while LSM-trees retain an advantage for read-heavy homogeneous access patterns. This transitional behavior indicates that TIDEHUNTER's advantages grow progressively as value sizes increase beyond this threshold.

6.2.3 Effect of Relocation. To evaluate relocation's impact on storage efficiency and performance, we ran a delete-heavy workload (100% deletes) under both uniform (θ=0) and skewed (θ=2) access patterns (Figure 9), after the database has been pre-filled with 1 TB of data. We used large values (1 KB). Without relocation, deleted entries leave gaps that accumulate, consuming 1148 GB and 1157 GB respectively. With relocation enabled, storage drops to 335 GB (uniform) and 933 GB (skewed)—a 71% and 19% reduction. The smaller savings under skew occur because only frequently accessed keys are deleted, leaving fewer reclaimable gaps. Throughput overhead remains minimal: 3% for uniform and 4% for skewed workloads.

workloads with high skew, TIDEHUNTER achieves 5.03M ops/s for get operations, surpassing RocksDB's 4.43M ops/s by 14%. For exists operations, TIDEHUNTER's advantage grows to 33% (5.31M vs 4.00M ops/s). This reversal occurs because TIDEHUNTER's in-memory 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.

Summary. For small 64-byte values with homogeneous access patterns, LSM-tree-based systems outperform TIDEHUNTER by 2–2.5×. However, TIDEHUNTER regains the advantage under skewed read workloads where caching effects dominate. This suggests that TIDEHUNTER is better suited for workloads with larger values or significant temporal locality, while LSM-trees remain preferable for small-value workloads with homogeneous access patterns.

The results for 128-byte values are in Figure 8. The 128-byte value size represents a transitional point between small and large value performance characteristics, where neither architecture holds a decisive advantage across all workloads.

Write Performance. Under write-only workloads (Figure 8a, left-most bars), all systems achieve comparable throughput: TIDEHUNTER at 437K ops/s, RocksDB at 430K ops/s, and BlobDB at 423K ops/s. This near-parity suggests that 128 bytes approaches the crossover point where TIDEHUNTER's single-write advantage begins to offset LSM-trees' efficient small-entry handling.

6.3 Index Format Comparison

We microbenchmark the lookup performance of TIDEHUNTER's two persistent index formats: the optimistic index and a naive baseline.

- **Header-based index**: This format uses a fixed-size header of 128 entries (1 KB total) to partition keys into micro-cells based on their prefix. Each header entry stores offsets pointing to a contiguous region of sorted key-value pairs. A lookup requires exactly two I/O operations: reading the 8-byte header entry to determine the data region, followed by reading and binary searching the data region.
- **Optimistic index** (Section 4.2): This format assumes a uniform key distribution to estimate the probable location of a key within the sorted index file. It uses a window-based search strategy with a default window size of 800 entries. The search begins at the estimated offset and iteratively moves the window until the target key falls within bounds, then performs a binary search. This approach eliminates header overhead and can achieve single-roundtrip lookups when the distribution estimate is accurate.

Data Generation. We generate 25 000 separate index files, each containing 1 000 000 randomly populated entries. Each entry consists of a 32-byte key generated uniformly at random and an 8-byte WAL position. The indices are serialized to disk using their respective formats, resulting in approximately 1 TB of data per format.

Benchmark Configuration. The benchmark performs 1 000 000 lookups per run across 10 independent runs. Lookups are grouped into batches of 1 000 operations for timing measurements. The window size for the optimistic index is set to 800 entries. We evaluate both single- and multi-threaded configurations, and optionally enable direct I/O to bypass the operating system's page cache.

Workload. The benchmark uses a random access pattern where each lookup selects a uniformly random index from the 25 000 available indices and queries a randomly generated 32-byte key. Since the lookup keys are generated independently of the keys stored in the indices, this workload consists predominantly of negative lookups (keys not present in the index), which represents a worst-case scenario for the optimistic index as it may require multiple window movements to determine key absence.

Results. Figure 10 shows the throughput scaling behavior of both index formats as a function of thread count. For the optimistic index, we use the optimal window size for each configuration (which varies from 1600 at low thread counts to 200–400 at high thread counts). For the header-based index, window size does not affect its performance.

With direct I/O enabled, the optimistic index outperforms the header-based index at all thread counts, achieving up to 194K lookups/s at 48 threads compared to 156K for the header-based index—a 24% improvement. Without direct I/O, both indices plateau at high thread counts due to page cache contention, with the optimistic index achieving 71K lookups/s compared to 57K for the header-based index.

Figure 10 shows the sensitivity of the optimistic index to window size across different thread counts. The header-based index maintains constant throughput regardless of window size (since it always performs exactly two I/O operations per lookup), while

the optimistic index exhibits a clear trade-off: smaller windows require more iterations but read less data per iteration, while larger windows reduce iterations but increase I/O volume. The optimal window size decreases as thread count increases, reflecting the benefits of smaller I/O operations under high concurrency.

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 to evaluate performance impact and demonstrate production readiness. We deployed Sui in a test cluster of 150 geographically distributed machines under an artificial load of 6,000 transactions per second.

All hosts started the experiment using RocksDB and at 12:30 switched to using TIDEHUNTER. These results are presented in Figures 11 and 12. 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 11 before 12:30). In contrast, once TIDEHUNTER is deployed the same hosts maintain stable latency throughout the test period (Figure 11, after 12:30). The same characteristics are also present in the disk I/O utilization (Figure 12), where the high write amplification of RocksDB results in 1-4GB/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 100MB/s for both reads and writes.

Since TIDEHUNTER sustained 6,000 TPS without any issue, we ran a second experiment with a load of 8,500 TPS. The performance of the same cluster using TIDEHUNTER under this increased load is demonstrated in Figure 13, where both throughput and latency remain steady during five hours of sustained load.

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

7 RELATED WORK

LSM-Tree Stores. LSM-trees [30] organize data into sorted runs across multiple levels, trading write amplification for read performance through background compaction. Stores like RocksDB [7, 17], LevelDB [14], and PebblesDB [36] have become industry standards, with various optimizations including lazy leveling [13], optimal Bloom filter allocation [12], and concurrent compaction [18]. Despite these improvements, compaction fundamentally rewrites data 5–10× across levels. TIDEHUNTER sidesteps this entirely: its append-only WAL eliminates multi-level compaction, achieving 8.4× higher write throughput than RocksDB.

Key-Value Separation. WiscKey [26] pioneered storing values in a separate log (vLog), dramatically reducing write amplification for large values. Production systems including BlobDB [27], Titan [35], and DiffKV [25] build on this approach, while HashKV [8] and BVLSM [24] optimize value placement and garbage collection. Kreon [34] uses memory-mapped KV separation for flash storage, while BadgerDB [16] implements WiscKey's design in Go. These systems still maintain an LSM-tree for keys, inheriting compaction
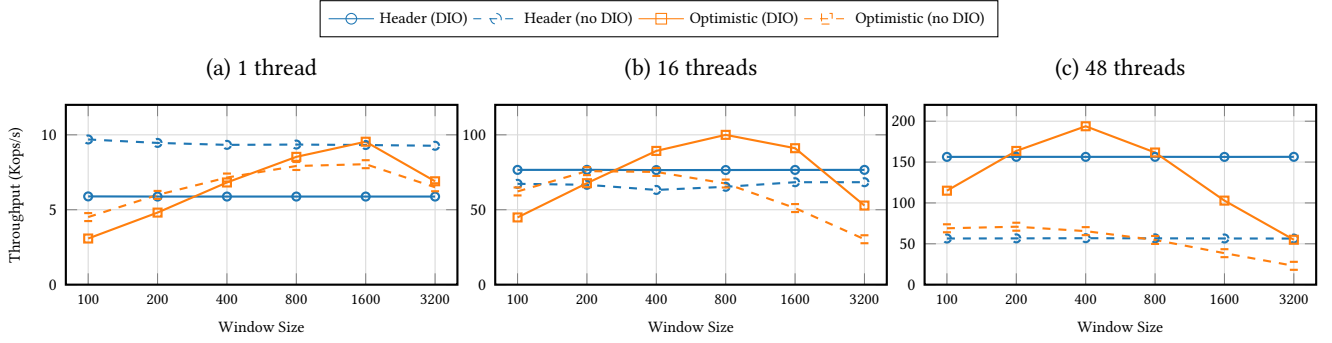
(a) 1 thread (b) 16 threads (c) 48 threads

Figure 10: Index lookup throughput vs. window size for different thread counts. The header-based index maintains constant throughput (flat lines), while the optimistic index performance varies with window size. The optimal window size decreases as concurrency increases.
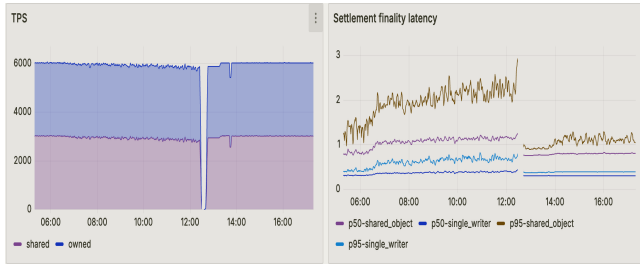

Figure 11: Transaction latency comparison on Sui blockchain under 6,000 TPS load. RocksDB (before 12:30) shows latency degradation over time. TIDEHUNTER (after 12:30) maintains stable latency.
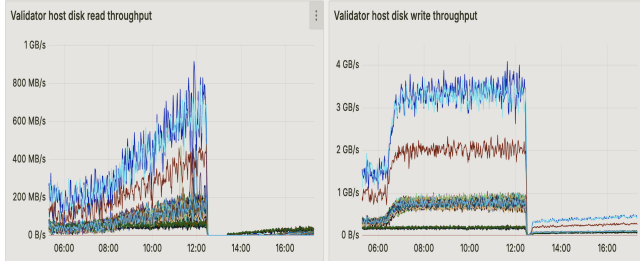

Figure 12: Disk I/O utilization comparison on Sui blockchain under 6000 TPS load. TIDEHUNTER (after 12:30) achieves significantly lower disk read and write rates than RocksDB (before 12:30).
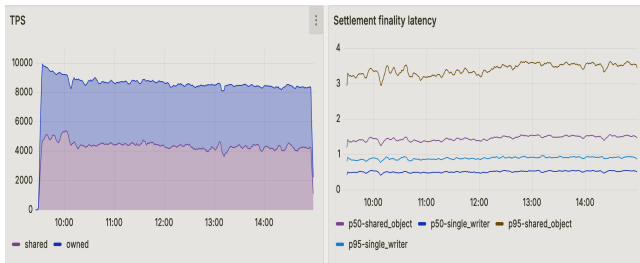

Figure 13: Performance of Sui using TIDEHUNTER under 8500 TPS load. Metrics remain steady, unlike RocksDB with 6000 TPS (Figure 11, before 12:30).

overhead. TIDEHUNTER differs fundamentally: the WAL serves as the unified value store with lazy-flushed index tables, avoiding both LSM compaction and separate vLog management.

B-Tree and Memory-Mapped Stores. LMDB [11] uses copy-on-write B+ trees with shadow paging, enabling zero-copy reads but limiting to single-writer concurrency. CedrusDB [38] combines memory-mapped lazy-tries with WAL, the most architecturally similar prior work to TIDEHUNTER. Traditional systems like BerkeleyDB [29] and SQLite [21] use B-trees with logging and locking, while redb [5] offers a modern Rust implementation. Unlike B-tree stores requiring in-place updates or copy-on-write, TIDEHUNTER's append-only design avoids page splits and tree rebalancing entirely.

Hardware-Optimized Stores. KVell [23] maximizes NVMe bandwidth via per-CPU slices without data sorting, sharing TIDEHUNTER's philosophy of avoiding sort overhead. SpanDB [10] places WAL on fast NVMe with SPDK for parallel I/O, while FASTER [9] uses epoch-protected hybrid logs achieving 160M ops/sec. SILK [2] and TRIAD [1] address LSM write stalls through I/O scheduling and tiered storage. TIDEHUNTER takes a different approach: rather than optimizing around LSM limitations, it eliminates them architecturally while using standard mmap and O_DIRECT interfaces.

8 CONCLUSION

TIDEHUNTER is a key-value store that eliminates value compaction by treating the WAL as permanent storage: values are written once and never moved, while small, lazily-flushed index tables map keys to WAL positions. On 1 KB values, TIDEHUNTER achieves 8.4× higher write throughput than RocksDB, 1.7× faster point queries, and 15.6× faster existence checks, with LSM-trees regaining advantage only below 128 bytes. Integration with Sui validators confirms these gains hold in production, maintaining stable performance under loads that collapse RocksDB-backed systems. The tradeoff is generality for efficiency on an increasingly common pattern: hash-keyed, kilobyte-scale, write-heavy workloads found in content-addressable storage, deduplication, and blockchain systems.

REFERENCES

[1] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *USENIX ATC*. USENIX ATC, Santa Clara, CA, 363–375.

[2] 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 ATC, Renton, WA, 445–460.

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

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

[5] Christopher Berner. 2023. redb: An Embedded Key-Value Database in Pure Rust. https://github.com/cberner/redb.

[6] Sam Blackshear, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Xun Li, Mark Logan, Ashok Menon, Todd Nowacki, Alberto Sonnino, et al. 2024. Sui lutris: A blockchain combining broadcast and consensus. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 2606–2620.

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

[8] 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 ATC, Boston, MA, 1007–1019.

[9] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *SIGMOD*. SIGMOD, Houston, TX, 275–290.

[10] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 2021. SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage. In *FAST*. FAST, Virtual, 17–32.

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

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

[13] 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*. SIGMOD, Houston, TX, 505–520.

[14] Jeffrey Dean and Sanjay Ghemawat. 2011. LevelDB. https://github.com/google/leveldb.

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

[16] Dgraph Labs. 2017. BadgerDB: Fast Key-Value DB in Go. https://github.com/dgraph-io/badger.

[17] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2021. RocksDB: Evolution of Development Priorities in a Key-Value Store Serving Large-Scale Applications. *ACM Transactions on Storage* 17, 4 (2021), 1–32.

[18] Robert Escriva, Bernard Wong, and Emin Gün Sirer. 2012. HyperDex: A Distributed, Searchable Key-Value Store. In *SIGCOMM*. SIGCOMM, Helsinki, Finland, 25–36.

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

[20] Haochen He, Erci Xu, Shanshan Li, Zhouyang Jia, Si Zheng, Yue Yu, Jun Ma, and Xiangke Liao. 2023. When Database Meets New Storage Devices: Understanding and Exposing Performance Mismatches via Configurations. *Proceedings of the VLDB Endowment* 16, 7 (2023), 1712–1725.

[21] D. Richard Hipp. 2000. SQLite. https://www.sqlite.org/.

[22] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka. 2009. Write Amplification Analysis in Flash-Based Solid State Drives. In *SYSTOR*. SYSTOR, Haifa, Israel, 1–9.

[23] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: The Design and Implementation of a Fast Persistent Key-Value Store. In *SOSP*. SOSP, Huntsville, ON, Canada, 447–461.

[24] Ming Li, Yu Zhang, Wendi Cheng, Wei Yuan, Jianjun Wang, Weimin Zheng, and Jidong Zhang. 2025. BVLSM: Write-Efficient LSM-Tree Storage via WAL-Time Key-Value Separation. arXiv:2506.04678.

[25] 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 ATC, Virtual, 673–687.

[26] 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*. FAST, Santa Clara, CA, 133–148.

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

[28] Meta. 2024. RocksDB: A Persistent Key-Value Store for Fast Storage. https://github.com/facebook/rocksdb.

[29] Michael A. Olson, Keith Bostic, and Margo I. Seltzer. 1999. Berkeley DB. In *USENIX ATC*. USENIX ATC, Monterey, CA, 183–192.

[30] 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.

[31] OpenMetal. 2024. OpenMetal: On-Demand Private Clouds. https://openmetal.io.

[32] Alexandru A. Ormenisan, Mahmoud Ismail, Fabio Buso, Joonseok Kim, Moritz Meister, and Jim Dowling. 2024. The Hopsworks Feature Store for Machine Learning. In *SIGMOD Companion*. SIGMOD Companion, Santiago, Chile, 601–604.

[33] 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.

[34] 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), 1–21.

[35] PingCAP. 2019. Titan: A RocksDB Plugin for Key-Value Separation. https://github.com/tikv/titan.

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

[37] Mendel Rosenblum and John K. Ousterhout. 1992. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems* 10, 1 (1992), 26–52.

[38] 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.

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