

TreePass: Lightweight Tree-Guided Cache Management for B^+-tree Key-Value Stores

Jin Yang, Hyeonseok Oh, Young-ri Choi
UNIST

Abstract

With the rise of NVMe SSDs, the performance gap between random and sequential I/O has narrowed, renewing interest in B^+-tree-based key-value stores (KVSs) for modern data systems. In larger-than-memory B^+-tree KVSs, performance critically depends on cache eviction decisions. However, existing page-cache eviction policies treat each page independently using only its own access history. As a result, they fail to exploit subtree-level hotness arising from key-range locality or preserve accumulated history across out-of-place updates, causing pages in hot subtrees to be misclassified and prematurely evicted. We observe that B^+-tree operations naturally expose lightweight, address-independent signals for subtree hotness and future page invalidation. Based on these insights, we propose **TreePass**, a lightweight caching system that augments CLOCK eviction with per-pivot counters embedded in the B^+-tree. With these counters, TreePass performs early eviction of cold-subtree pages and differentiated reference-counter initialization for newly created pages, suppressing unnecessary writebacks without additional metadata. We implement TreePass in SplinterDB and evaluate it across YCSB and real-world workloads. Across all evaluated workloads, TreePass achieves up to 101% higher throughput, 70% fewer page writes, and 15% lower P99 tail latency than state-of-the-art caching policies.

1 Introduction

Key-value stores (KVSs) are a core building block of modern data management, widely used in cloud services, distributed databases, and caching infrastructures [5, 11, 13, 15]. While LSM-tree-based designs have long dominated write-intensive workloads, the rise of NVMe SSDs has significantly reduced the performance gap between random and sequential I/O [16, 44]. This shift has led to renewed interest in B^+-tree-based storage systems, which can now provide competitive performance on modern storage hardware [7, 8, 17, 25, 30, 44].

Larger-than-memory B^+-tree KVSs commonly rely on an in-memory page cache that mediates disk accesses at the granularity of fixed-size pages [8, 25, 30], and its eviction decisions strongly influence system throughput. However, accurate eviction is challenging due to two key properties of these systems. First, real-world production workloads exhibit strong key-range locality, where accesses concentrate within a small number of contiguous key ranges [4]. In a B^+-tree, this translates into *subtree-level hotness skew*, a property that is not visible from page-granularity tracking alone. Second, modern B^+-tree KVSs heavily rely on *out-of-place updates* (e.g., compaction, copy-on-write, and node splits), which rewrite pages to new locations and break the assumption of stable page addresses—that repeated accesses to the same data map to a fixed physical page. Consequently, newly created pages lose accumulated access history, while the resulting writebacks compete with foreground reads for I/O bandwidth, particularly under read-write mixed workloads.

Prior work improves eviction accuracy using auxiliary history structures such as ghost queues [12, 19, 20, 29] or learned models [35–37, 40, 46]. Lightweight eviction algorithms such as SIEVE [45] and S3-FIFO [43] further reduce overhead. However, when applied to a user-level page cache, all these approaches operate at page granularity: each page maintains its own access history, and eviction decisions rely on stable page addresses. Consequently, they neither exploit subtree-level hotness arising from key-range locality nor preserve accumulated access history across out-of-place updates, leading to premature eviction of hot pages and unnecessary writebacks.

We observe that the B^+-tree itself provides lightweight signals to address these challenges. During traversal, internal-node pivots (separator keys) are accessed to route queries, and their access frequency reflects the hotness of the corresponding subtrees, while how frequently a key range is rewritten indicates which pages are likely to be invalidated soon. Because these signals reside in the tree's existing pivots, they need no separate metadata structures and are updated as part of normal tree operations. Based on these insights, we propose **TreePass**, a lightweight caching system that enhances CLOCK eviction with subtree-level structural signals. TreePass embeds small per-pivot counters into existing B^+-tree

nodes, requiring no additional global metadata structures, and leverages them during tree operations to guide eviction decisions. It enables early eviction of cold-subtree pages by identifying pages unlikely to be re-accessed before the next eviction cycle, preserves cache history across out-of-place updates, and suppresses unnecessary writebacks for pages likely to be invalidated.

The contributions of this paper are as follows. First, we identify two key limitations of existing cache eviction policies in B^+-tree KVSs: (i) the inability of page-level policies to exploit subtree-level locality in production workloads, and (ii) the reliance on address-based access history, which becomes ineffective under out-of-place updates as rewritten pages are treated as new objects. Second, we show that the B^+-tree structure yields two lightweight, address-independent signals for subtree hotness and future page invalidation. Third, we design and implement TreePass [1] in SplinterDB [8], a high-performance B^+-tree KVS for NVMe SSDs, and evaluate it on both YCSB workloads with diverse access distributions and a real-world workload derived from a Twitter cache trace [42]. TreePass consistently improves performance by reducing both page reads and unnecessary page writes. Across all evaluated workloads, it achieves up to 83% higher throughput, 70% fewer page writes, and 23% lower P99 tail latency than CLOCK, and up to 101% higher throughput, 70% fewer page writes, and 15% lower P99 tail latency than S3-FIFO. The largest improvements occur under read-write mixed workloads.

2 Background and Motivation

2.1 SplinterDB

SplinterDB [8] is a B^+-tree-based key-value store (KVS) designed for NVMe SSDs. It employs the STB$^\varepsilon$-tree structure, as shown in Figure 1, and achieves superior insertion and point query throughput compared to RocksDB [7, 8].

STB$^\varepsilon$-tree structure. The STB$^\varepsilon$-tree is a "tree-of-trees" structure consisting of a *trunk tree* that forms the overall backbone and multiple *branch trees* that store the actual key-value pairs. Each trunk node maintains pivot keys (also known as separator keys) and child pointers for its children, along with pointers to the branches associated with the node. Each branch tree is a B^+-tree whose internal nodes maintain pivot keys and child addresses, and whose leaf nodes store the actual key-value pairs. Actual data resides in the branches; trunk nodes themselves contain only metadata. During queries, a quotient maplet [7] is used as a filter to determine which branches may contain a given key.

Query path. A point lookup first searches the memtable. If the key is not found, the trunk tree is traversed from root to leaf. At each trunk node, the pivots are searched to determine the correct child, and the quotient maplet is consulted to identify the branches that may contain the key [7]. Only those branches are selectively searched. Throughout this process, trunk node pages, maplet pages, and branch node pages are all accessed through SplinterDB's user-level page cache, and the pivot access frequency directly reflects the access frequency of the corresponding subtree.

Insertion and compaction. Insertions are buffered in an in-memory memtable. When the memtable is full, it is converted into a new branch (B^+-tree) and added to the root trunk node. When a trunk node becomes full, it first flushes the references to the active branches to its child, then merges those branches into a single new branch through compaction (flush-then-compact [7, 8]). Compaction reads pages from existing branches and produces a new branch, creating a large number of new pages while invalidating old ones. This process repeats recursively down the tree, and trunk node splits generate additional new pages. Updates and deletes follow the same path: both are buffered in the memtable—a delete as a tombstone—and applied lazily during compaction.

Cache. SplinterDB operates a single user-level page cache through which all data accesses are routed [8]. The cache manages fixed-size 4 KB pages and stores all page types: trunk node pages, branch node pages, maplet pages, and memtable pages. Memtable pages are allocated from the cache and kept resident until the memtable is flushed. The cache uses a fixed-size ring buffer with CLOCK-based eviction. SplinterDB achieves concurrency through thread-local eviction hands and distributed reader-writer locks. A single point lookup involves trunk node accesses proportional to the trunk height, plus maplet and branch accesses at each level, so the cache's eviction decisions directly impact query performance.

2.2 Cache in Key-Value Stores

In-memory caches are a key component for reducing disk I/O in KVS. The underlying data structure of a cache can be broadly classified into linked-list-based and ring-buffer-based designs. Linked-list-based designs are used by LRU, ARC [29], 2Q [20], SIEVE [45], and others. LRU-style policies update list pointers on every access, while SIEVE performs lazy promotion, updating only a visited bit on hits and manipulating list pointers only during insertion and eviction; such pointer operations cause lock contention under concurrent workloads. Ring-buffer-based designs, in contrast, only update per-entry metadata atomically on access and require no entry movement, resulting in lower lock contention. Due to these advantages, recent KVS implementations increasingly favor ring-buffer-based caches over linked-list-based designs [8, 13].

A representative eviction policy that operates on a ring buffer is CLOCK [10]. CLOCK maintains a reference bit per entry; the eviction hand sweeps through entries and replaces those whose bit is 0. Accessed entries have their reference bit set to 1, which is cleared to 0 when the hand passes. Multi-bit variants, known as *generalized CLOCK* (GCLOCK) [31], replace the reference bit with a reference counter that is incre-

[1] https://github.com/unist-ssl/TreePass

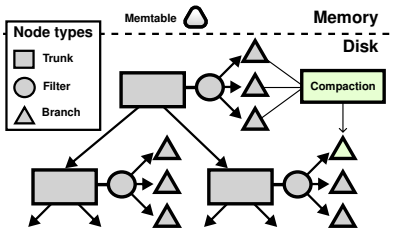


Figure 1: STB$^{\varepsilon}$-tree structure in SplinterDB

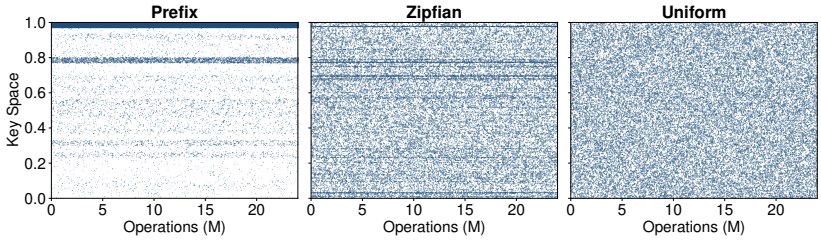


Figure 2: Leaf-level access patterns under three distributions

mented on hits and decremented as the hand passes, evicting entries that reach 0. Since the hit path requires only a single atomic increment, the per-operation overhead is very low. However, CLOCK cannot promptly evict entries that are accessed only once and never reused (i.e., *one-hit wonders* [43]). Such entries occupy the cache until the eviction hand reaches them, which can increase the miss ratio.

The recently proposed S3-FIFO [43] addresses this problem through quick demotion—a strategy that promptly evicts entries that are not re-accessed after insertion [41]. S3-FIFO is a general-purpose eviction algorithm designed for diverse environments including KVS, Content Delivery Network (CDN), and block caches, and consists of three FIFO queues: Small, Main, and Ghost. Newly inserted entries first enter the small queue; if they are not re-accessed, they are quickly evicted (quick demotion). Only re-accessed entries are promoted to the main queue. Fingerprints of entries evicted from the small queue without re-access are recorded in the ghost queue; if such an entry is requested again in the future, it is inserted directly into the main queue. Through this design, S3-FIFO achieves state-of-the-art miss ratios while maintaining the high scalability of FIFO-based designs [43].

In this paper, we adopt CLOCK as the base structure of our proposed TreePass because it requires no additional metadata structures, such as ghost queues, allowing TreePass to fully leverage the advantages of the ring buffer. TreePass extends the original 1-bit reference bit to a 3-bit reference counter to differentiate eviction priority more precisely based on access frequency. We also implement S3-FIFO on SplinterDB's ring-buffer-based cache and use it as a baseline for comparison.

2.3 Issues of Cache Management

In larger-than-memory KVSs, KV pairs are stored on disk while memory is used to cache data and index nodes. As a result, every cache miss triggers a disk read that costs tens of microseconds on NVMe SSDs—orders of magnitude slower than an in-memory access [16, 25]. Under concurrent workloads, each outstanding read occupies an NVMe queue slot and consumes I/O bandwidth; as the device approaches saturation, queueing delays further inflate the per-miss cost [16]. In read-write mixed workloads, dirty-page writebacks issued during eviction compete for the same bandwidth, increasing read latency and degrading throughput [25, 27]. Consequently, reducing cache misses directly decreases the number of disk reads, while suppressing unnecessary writebacks frees I/O bandwidth for foreground queries [27, 39].

Workload Characteristics. To understand how cache misses can be reduced, we examine the access patterns of real workloads in the context of the B^+-tree structure. While prior studies often model workloads using uniform or Zipfian distributions, such models capture access frequency skew but fail to reflect how hot data is spatially distributed across the key space. Recent studies show that production key-value workloads exhibit strong key-range locality: hot keys are not scattered randomly but concentrated within a small number of contiguous key ranges [4]. MixGraph [4], derived from Facebook's ZippyDB traces, introduces a prefix-based distribution (Prefix-dist) that partitions the key space into hot and cold ranges and directs more requests to the hot ranges [21], closely matching production access patterns. Several subsequent system studies adopt this distribution as a representative production-like workload for evaluating KVS performance [3, 14, 21, 22, 24]. These studies suggest that realistic cache behavior must be understood not only through frequency skew but also through spatial clustering of accesses within the key space. Motivated by this, we analyze three representative distributions—uniform, Zipfian, and prefix-based—to understand how key-range locality interacts with the B^+-tree structure.

Observation #1: In real workloads, accesses are skewed at the subtree level, and the B^+-tree structure already contains this information. Figure 2 visualizes the key space distribution of branch leaf lookups across operations under the three access distributions on a read-only workload (YCSB-C; detailed setup in §4). Each point represents a sampled leaf lookup; the x-axis is the number of operations, and the y-axis is the normalized position within the key space. Under the prefix distribution that models production workloads (MixGraph [4]), accesses concentrate on specific key ranges, forming distinct hot bands while the remaining regions are rarely accessed. Under the Zipfian distribution, accesses exhibit frequency skew where a small set of keys are accessed much more frequently than others, but these hot keys are scattered across the key space rather than forming contiguous hot regions. In contrast, under the uniform distribution, accesses are spread evenly across the entire key space, with no hot/cold distinction.

In a B^+-tree, key ranges naturally correspond to subtrees: pages serving the same key range belong to the same subtree, so key-range level locality directly translates into subtree level hotness skew. This information is already observable in the

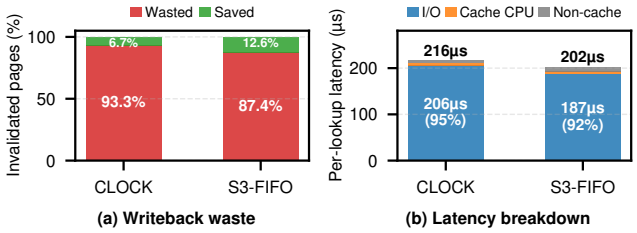


Figure 3: Writeback waste and its performance impact

tree's internal nodes. Each lookup repeatedly accesses the pivots (separator keys) that route the query to the correct child subtree; therefore, the access frequency of a pivot naturally reflects the hotness of the corresponding subtree. This signal is collected naturally during tree traversal and requires no additional metadata structure, as it is tied to the tree structure rather than to physical page addresses.

However, existing cache management policies such as CLOCK and S3-FIFO make eviction decisions solely based on per-page access history. As a result, they fail to exploit this structural signal. Even when accesses repeatedly concentrate on a small number of hot subtrees, pages belonging to the same subtree are treated as unrelated objects, preventing the cache from recognizing and preserving hot regions of the tree.

Impact of Out-of-Place Updates. B^+-tree KVSs frequently create pages at new addresses through out-of-place updates such as compaction, copy-on-write, and node splits, introducing an additional challenge for cache management. Out-of-place update—writing a modified page to a new disk location—is widely used in B^+-tree-based storage systems and several KVSs due to its benefits for crash recovery and snapshot/MVCC support [1, 6, 34]. Many modern B^+-tree KVSs rely on background compaction and copy-on-write mechanisms, frequently rewriting pages at new addresses during normal operation. Therefore, write amplification remains a central optimization target on modern storage hardware [2, 23, 32]. In SplinterDB, for example, compaction, copy-on-write, and splits of trunk nodes, and memtable node splits all produce out-of-place updates. From a cache management perspective, this approach entails two side effects.

First, existing eviction algorithms [10, 18, 29, 43, 45] assume that repeated accesses to the same data always arrive at the same page address. Under out-of-place updates, however, the modified page is written to a new address, breaking the link between the access history recorded for the old address and the new one. For example, in S3-FIFO, the new address has no link to the old one and does not match fingerprints recorded in the ghost queues. The newly created page therefore appears as a cold object to the cache, even if it belongs to a frequently accessed subtree. As a result, pages generated in hot subtrees are often misidentified as cold and evicted prematurely, causing additional cache misses and disk reads.

Second, compaction frequently generates dirty pages that will soon be invalidated by subsequent compaction. Existing eviction policies cannot anticipate this future invalidation. Because dirty pages must be written back before eviction, this can trigger unnecessary writebacks for pages whose contents will soon become obsolete. For example, S3-FIFO's small queue evicts compaction-generated pages quickly before they are re-accessed, making such writebacks even more frequent. In SplinterDB, compaction reads existing branches and produces a new branch, thereby invalidating the old pages. If a dirty page has already been written back before it is invalidated, the I/O spent on that writeback is entirely wasted.

Observation #2: A large fraction of writebacks are wasted on pages that are soon invalidated by compaction. Figure 3 quantifies this effect under a read-write mixed workload (YCSB-A, prefix distribution, 1 GB cache; detailed setup in §4) using CLOCK and S3-FIFO. Figure 3(a) shows the fraction of branch node pages that had already been written back before they were invalidated by compaction. With CLOCK, 93.3% of invalidated branch node pages had already been written back; S3-FIFO shows a similar trend at 87.4%. These writebacks are ultimately unnecessary, as the written-back contents are later superseded by compaction.

Figure 3(b) breaks down average per-lookup latency under the same workload into three components: I/O time (disk read stalls on cache misses), cache CPU time (cache-internal operations such as eviction decisions and reference counter updates), and non-cache CPU time (the KVS's own operations such as tree traversal and filter lookup). I/O time dominates overwhelmingly—95.3% of the total latency for CLOCK and 92.4% for S3-FIFO—leaving cache and non-cache CPU overhead as minor components. In such an I/O-dominated regime, unnecessary writebacks compete with foreground reads for limited I/O bandwidth, amplifying the impact of the waste shown in Figure 3(a). Suppressing these futile writebacks is therefore a direct path to improving throughput in read-write mixed workloads [25, 27].

3 Design

3.1 Overview

TreePass is a caching system built on top of SplinterDB's CLOCK-based cache (§2.2) that incorporates two structural signals from the STB$^\varepsilon$-tree: the access count of each subtree, tracked via pivots in branch internal nodes, and the compaction count of each key range, tracked via trunk node pivots.

As discussed in §2.3, B^+-tree KVSs exhibit two key properties: strong key-range locality, which manifests as subtree-level hotness skew (Observation #1), and out-of-place updates, which invalidate accumulated access history and waste writebacks (Observation #2). TreePass addresses both through a single design intuition: tying cache-management signals to the tree structure rather than to page addresses. Concretely, TreePass embeds two counters in pivots. Together, they (i) capture subtree-level access and update patterns through tree traversal and compaction, and (ii) are tied to the tree structure rather than to page addresses. TreePass exploits these signals

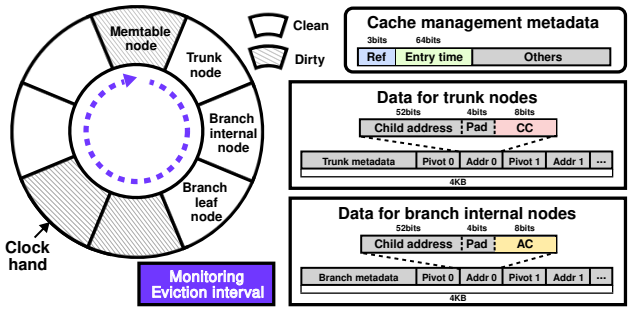


Figure 4: Overall architecture of TreePass

through two mechanisms. First, it leverages the per-subtree access counts to identify cold pages early (§3.2). Second, for pages created by out-of-place updates, it assigns differentiated initial reference counters based on tree-structural information—preserving eviction priority despite history discontinuity and reducing unnecessary writebacks (§3.3). In both mechanisms, the tree passes its signals to the cache—hence the name *TreePass*.

As TreePass targets a dataset larger than memory, its tree nodes reside on disk and are brought into the cache on demand. Figure 4 shows the overall structure of TreePass's cache. The cache is organized as a 3-bit CLOCK-based ring buffer. Each cache entry consists of cache management metadata and a data page. The metadata fields—including the reference counter, page address, dirty flag, node type, and fields for page I/O and concurrency control—are inherited from SplinterDB's original CLOCK cache design. However, TreePass extends the reference counter from 1 bit to 3 bits. The wider counter provides additional initialization states, enabling context-dependent initial priorities for newly inserted pages (§3.3). In addition, TreePass adds an 8-byte time field to record when the page was loaded into the cache. The same timestamp serves two purposes: branch internal nodes use this time field to determine how frequently each child subtree is accessed (§3.2), while trunk nodes use it to determine how frequently compaction occurs for each key range (§3.3).

TreePass introduces *Tree Counter*s, a new counter type embedded in the unused byte of the child address field of existing pivot entries, requiring no additional memory. A Tree Counter serves different roles depending on the node type. For branch internal nodes, it acts as an *Access Counter* (AC), tracking the access count of the corresponding child subtree. It uses 7 bits as an access counter and 1 bit as a cache-resident flag (§3.3). For trunk nodes, it acts as a *Compaction Counter* (CC), tracking the number of compactions performed for the corresponding key range. In this case, all 8 bits are used as a compaction counter. When AC or CC approaches saturation, all sibling counters in the same node are reset and the entry time is re-anchored to restart the estimation window. Upon loading a page from disk, both counters are reset and the entry time is initialized to the current time. For branch leaf nodes in the cache, their entries also include cache management metadata, while the data itself consists of key-value pairs without any additional fields.

As shown in §2.3, real workloads exhibit subtree-level access skew that existing page-level eviction policies cannot exploit. In a B^+-tree, every access to a page in a subtree must pass through the parent pivot that routes to that subtree. A hot subtree therefore drives frequent accesses to its parent pivot, yielding a short re-access interval, while a cold subtree yields a long one. TreePass exploits this relationship by comparing two intervals: the **Eviction Interval (EI)** and the **Re-access Interval (RI)**. EI is the time it takes for the CLOCK hand to complete one full sweep of the cache. If a page is not re-accessed within this interval after being loaded into the cache, it becomes a replacement candidate when the eviction hand reaches it. EI thus represents the maximum re-access interval tolerated for a page to remain in the cache: when the cache is full and eviction is frequent, sweeps complete faster and EI decreases; when there is spare capacity, sweeps slow down and EI increases. RI is an estimate of the average re-access interval for a given page. From this information, how frequently the page is accessed can be inferred.

TreePass exploits the subtree-level hotness signal (i.e., AC) to proactively identify and remove cold pages during tree traversal. We call this mechanism **early eviction**. Upon a cache miss to a branch node page, TreePass estimates the page's RI from its parent pivot entry's AC and compares it with EI. If RI $\leq$ EI, the page is inferred to belong to a sufficiently hot subtree, classified as *warm*, and retained in the cache; otherwise, it is classified as *cold* and evicted immediately, without waiting for the CLOCK hand to reach it (§3.2). Early eviction complements rather than replaces CLOCK's existing mechanism. Cache-resident pages continue to be managed solely by CLOCK at page granularity, while subtree-level information is consulted only when a page is fetched on a cache miss. Importantly, the subtree serves only as the *source* of the hotness signal–not as the unit of caching or eviction. Every page is admitted, retained, and evicted individually. This design enables TreePass to exploit subtree traversal frequency, which is unavailable from page-granularity tracking alone, to proactively identify pages from cold subtrees, thereby reducing wasted cache space while preserving standard page-granularity cache management.

Early eviction is applied only to branch node pages, which constitute the majority of the data. Trunk nodes contain only metadata about the branches and filters, and so use negligible DRAM [8]; they naturally reside in the cache and are therefore not subject to early eviction. Filter (maplet) pages are consulted during queries but are not traversed through a parent–child pivot relationship, so RI cannot be estimated for them; they are managed solely by the CLOCK reference counter. Memtable nodes reside almost entirely in DRAM [8] and are exempt from eviction, consistent with SplinterDB's original design.

EI is estimated by tracking the CLOCK hand's sweep

duration using an exponentially weighted moving average (EWMA). After each completed sweep, the sweep time T_{sweep} is measured and EI is updated as $\text{EI}_{\text{new}} = \alpha \cdot T_{\text{sweep}} + (1-\alpha) \cdot \text{EI}_{\text{old}}$, where α controls the responsiveness to recent changes. Early eviction requires a valid EI estimate. Until the first sweep completes, the cache is not yet full and EI is not available; during this period, the cache operates as standard CLOCK. Upon the first sweep completion, EI is initialized proportionally to the cache size (1 second per GB), as the first sweep time reflects cache warm-up conditions rather than steady-state eviction pressure. We find performance to be insensitive to both the smoothing factor α and the initial EI value (§4.9); accordingly, both use fixed system defaults.

RI is leveraged only for non-resident pages, as for cache-resident pages, the reference counter already tracks access count and manages eviction priority. For non-resident pages, the AC at each pivot within a branch internal node tracks the access count of the corresponding child subtree. Therefore, the RI of a page can be computed from the cache residency time of its parent page and the AC value at its corresponding pivot; the detailed computation is described in §3.2. Importantly, neither RI nor EI requires user configuration.

Note that, except during the initialization phase, a page may be early evicted multiple times until RI $\leq$ EI before it can be classified as warm. Consequently, only frequently accessed hot pages remain in the cache in TreePass. As shown in §4.5, this conservative criterion, which caches only "actual" hot pages, improves performance.

The discussion above concerns the criterion for classifying a page as warm. A separate design choice is when a page classified as cold should actually be evicted. Our current implementation evicts such pages immediately, which is effective when subtree hotness is relatively stable, as pages classified as cold are unlikely to be re-accessed within EI. When subtree hotness shifts over time, however, briefly deferring eviction may allow some pages to be reclassified as warm before being discarded, reducing unnecessary reloads. Designing workload-aware early eviction timing remains an opportunity for future work.

When a page is classified as warm and inserted into the cache, TreePass assigns it a differentiated initial reference counter value. A page read from disk during a query (i.e., a *clean* page) requires no writeback upon eviction and therefore receives a low initial value of 0. In contrast, a newly created dirty page receives a higher initial value to account for the writeback overhead (§3.3). Dirty pages created by trunk node copy-on-write or splits have a one-to-one correspondence with their predecessor pages, allowing their access history to be preserved by inheriting or distributing the reference counter accordingly. However, dirty pages created by branch compaction do not have such correspondence because compaction merges pages from multiple existing branches into new pages. In this case, the initial reference counter is determined using the reference counters of the input pages and the **Compaction Interval (CI)**—an estimate of the compaction interval for the key range covered by each trunk pivot (§3.3).

A dirty page created by compaction may soon be invalidated by a subsequent compaction, making any intervening writeback entirely wasted. If CI $\leq$ EI, the key range is likely to be compacted again before the page is evicted, so a high initial value is assigned to prevent unnecessary writeback. Similarly, if the input pages have high reference counters, the output page is likely to be frequently accessed and receives a high initial value.

3.2 Query

This section describes how TreePass leverages the subtree-level hotness signal identified in §2.3 to guide per-page eviction decisions during query processing. To locate a key-value pair, a query traverses the trunk tree from the root to a leaf. At each trunk node, it consults the quotient maplet to identify candidate branches and searches the corresponding branch trees. If the key is not found, the traversal proceeds to the appropriate child trunk node based on the pivot range. During branch tree traversal, each pivot's AC serves as a subtree hotness signal for the corresponding child subtree. Whether the AC is updated depends on the cache residency of the selected child page. If the child page is already in the cache (i.e., cache hit), its reference counter is incremented to protect it from eviction, and the corresponding AC update is skipped.

On the other hand, if the child page is not in the cache (i.e., cache miss), TreePass estimates the RI of the corresponding subtree and compares it with EI to classify the page as warm or cold. RI is computed using the AC value at the parent pivot and the cache residency time of the parent page. Specifically, RI is estimated as the parent page's cache residency time divided by the AC value:

$$\text{RI} = \frac{T_{\text{cur}} - T_{\text{entry}}}{\text{AC}} \tag{1}$$

where T_{entry} is the time the parent page was loaded into the cache, T_{cur} is the current time, and AC is the access counter maintained at the *parent pivot* that routes to the child page. Since AC is maintained per pivot rather than per page, RI estimates the average re-access interval of the specific child subtree. AC is read before the current access is reflected, so RI is computed using the counter value prior to incrementing.

If RI $>$ EI, the page is classified as cold and early evicted, and the corresponding AC is incremented by one. If RI $\leq$ EI, the page is classified as warm and retained in the cache, with its reference counter initialized to 0 since it is a clean page and requires no writeback, and the cache-resident flag in the AC is set to indicate that this child was recently cache-resident. Note that the early eviction decision for a child page is made at the parent node because it holds all the information needed to estimate the child's RI.

Figure 5 illustrates how TreePass classifies pages as warm or cold and how AC is tracked during query traversal. For simplicity, each internal node has a fanout of two, only internal

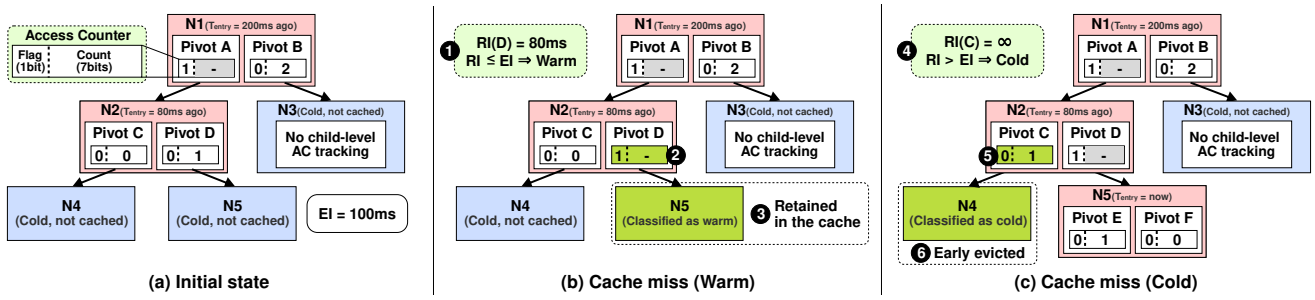


Figure 5: AC-based cache decisions during query traversal

nodes are shown, and EI is fixed at 100 ms throughout the example. In the initial state shown in Figure 5(a), internal node N1 has been cached for 200 ms and has two pivots. N1's Pivot A, which routes to child N2, has its cache-resident flag set (indicating N2 is in the cache), so AC tracking is skipped; N2 has been cached for 80 ms, with Pivot C (AC = 0) and Pivot D (AC = 1). N1's Pivot B, which routes to child N3, has AC = 2; N3 is not cached, so its child-level access counts cannot be tracked. In Figure 5(b), a query looks up a key whose search path passes through N1, N2, and N5 down to a leaf node. While traversing N2, TreePass finds that the child page N5 is not in the cache. Before descending, TreePass computes RI using the current AC value at Pivot D (AC = 1 as shown in Figure 5(a), not yet incremented for this access): RI = 80/1 = 80 ms. Since RI ≤ EI, the page is classified as warm (❶). The cache-resident flag at Pivot D is set (❷), and N5 is retained in the cache (❸). The query then descends into N5 toward its leaf. In Figure 5(c), N5—retained and traversed during the previous lookup—is now cached, and the AC at its Pivot E has been incremented to 1. Next, consider a subsequent query whose search path passes through N1, N2, and N4 down to a leaf node. While traversing N2, TreePass finds that the child page N4 is not in the cache. Since AC = 0, RI = ∞, which exceeds EI, so the page is classified as cold (❹). The AC at Pivot C is incremented to 1 (❺), and N4 is early evicted (❻).

While a child page remains in the cache, the AC at the corresponding pivot is not updated. If the page is later evicted and subsequently accessed again, a cache miss will occur. In this case, the cache-resident flag is set to 1, and the counter value may be lower than the actual access count. TreePass then clears the cache-resident flag and compensates for the AC's underestimation so that the estimated RI is slightly above EI:

$$\mathrm{AC}' = \max\left(\mathrm{AC},\ \left\lfloor \frac{T_{\mathrm{cur}} - T_{\mathrm{entry}}}{\mathrm{EI}} \right\rfloor - 1\right) \tag{2}$$

where $T_{\mathrm{cur}} - T_{\mathrm{entry}}$ is the parent page's cache residency time. The compensation is applied only if it increases the AC; otherwise, the existing AC is left unchanged. The resulting value AC' is capped at $2^7 - 1$, the maximum representable counter value. As a result, the page is classified as cold at this access, but if it is accessed again soon, it will likely be classified as warm. This helps mitigate misclassification caused by skipped counter updates while the page was resident in the cache. Recall that when AC reaches its maximum value, TreePass resets the counter and updates the entry time to the current time, restarting the counting window. Importantly, the cache-resident flag is preserved during this reset so that the compensation logic (Equation 2) is applied correctly afterward.

The overhead of AC maintenance is naturally bounded. On hot paths, most children are cache-resident, so AC updates are skipped, reducing overhead. When a cache miss does occur, AC is updated, regardless of whether the page is classified as warm or cold: a cold page increments the access counter, whereas a warm page sets the cache-resident flag. If a child page lies off the hot path and is not cached, the ACs at the pivot entries of that page are never updated, so counters in cold subtrees are not tracked unnecessarily.

3.3 Insertion and Compaction

This section describes how TreePass addresses the cache history disruption and writeback waste caused by out-of-place updates, as analyzed in §2.3. Insertions are buffered in an in-memory memtable (§2.1). When the memtable becomes full, it is converted into a new branch (B^+-tree) together with a quotient filter and attached to the root trunk node [8]. The resulting branch and filter pages enter the cache as dirty pages, and their reference counters are initialized to at least 1 to prevent immediate replacement by the CLOCK hand. When additional information is available—such as correspondence to a predecessor page or the compaction frequency—a higher initial value may be assigned.

As described in §2.1, branches accumulate in a trunk node until the node becomes full, triggering flush-then-compact [8]. In the flush phase, the parent's branch references are transferred to the child; in the subsequent compaction phase, the child merges the received branches into a single new branch. During compaction, key-value pairs from the input branches are read and merged in globally sorted order to produce the new branch. This process reads pages from existing branches and produces a large number of new branch node pages. TreePass determines the reference counter of each new branch node page using the following two signals.

First, the reference counters of input pages read during compaction are utilized. When a new output leaf is created,

its reference counter is initialized to the average reference counter of the input leaves merged into it. In contrast, the reference counter of an internal node is initialized to the maximum reference counter among its direct children—since every query to any child must pass through it, its access frequency is at least as high as that of its hottest child.

Second, CI is estimated from the CC recorded at each trunk node pivot and compared with EI. The counter is incremented after each completed compaction. CI is computed as the elapsed time since the trunk node entered the cache, divided by the compaction counter:

$$\mathrm{CI} = \frac{T_{\mathrm{cur}} - T_{\mathrm{entry}}}{\mathrm{CC}} \tag{3}$$

If CI $\leq$ EI, compaction is likely to occur again before the page is evicted, so the reference counter is set to the maximum value to prevent unnecessary writeback of pages that will soon become invalid. If CI $>$ EI, the default value of 1 is assigned. Note that TreePass only adjusts the timing of dirty-page writeback. Since SplinterDB's eviction policy already determines writeback timing without strict ordering guarantees, this introduces no additional crash-consistency risk; we discuss durability and crash recovery further in §5.2. The final reference counter is the larger of the two signals. The input page reference counter reflects how frequently the page was accessed, while the CC reflects how frequently compaction occurs. By considering both access frequency and update frequency, the initial value accounts for both dimensions.

Initial values based on CI are also applied to quotient filter pages generated during branch compaction. Since filters are built from fingerprints rather than by reading and merging existing filter pages, inheriting reference counters from input pages is not possible; thus, CI alone determines their initial value.

The above mechanism for assigning initial reference counter values requires that the entry time remains consistent as the trunk node undergoes structural modifications. Since SplinterDB manages trunk nodes via copy-on-write, every modification—including flush, compaction, and metadata updates—produces a new page at a new address. TreePass inherits the entry time from the old trunk node page to the new trunk node page whenever such a copy-on-write occurs. Similarly, when a trunk node split occurs, each new child inherits the entry time from the split source. This ensures that the time window used for CI estimation remains continuous regardless of structural changes in the trunk tree.

For trunk nodes, unlike branch node pages produced during compaction, there are cases where a one-to-one correspondence exists between old and new pages. During a flush, the trunk node itself is also modified as the branch list changes and pivot metadata is updated. Since trunk nodes are managed via copy-on-write, such modifications always produce a new page. In this case, the reference counter of the old trunk node page is directly inherited by the new page. If a trunk node split occurs after the flush due to excessive fanout, the node is divided into multiple children, and both the reference counter and the CC are evenly distributed among them. This ensures that CI can be correctly estimated when compaction subsequently occurs at each of the resulting child trunk nodes.

The elevated reference counter also affects CLOCK's write-back decisions. TreePass defers writeback for dirty pages whose reference counter is greater than 0; consequently, pages assigned a high initial value effectively have their writeback delayed.

4 Evaluation

4.1 Experimental Setup

Experimental setup. All experiments are conducted on a server with two Intel Xeon Gold 6526Y processors, providing 32 cores and 64 hardware threads in total, 314 GB of DDR5 DRAM (4800 MT/s, ECC), and a Samsung PM9A3 1.92 TB NVMe SSD (PCIe 4.0 $\times$4). The system runs Ubuntu 22.04.2 LTS with kernel 5.15.0.

We implemented TreePass on top of SplinterDB [8, 38] (commit `2b6d030`) by extending the ring-buffer cache to maintain ACs and CCs in pivot entries, perform early eviction during branch tree traversal, and assign differentiated initial reference counters to pages created by out-of-place updates. To minimize overhead, the entry time field is maintained only for branch and trunk node pages, which require it for warm/cold classification and CI tracking. For foreground queries, the timestamp is obtained once at the query entry point and reused throughout via a thread-local variable, avoiding repeated `platform_get_timestamp()` calls during cache accesses. We also implemented CLOCK-2, CLOCK-3, and S3-FIFO as caching policies on the same ring-buffer cache. SplinterDB is configured with direct I/O (O_DIRECT), reflecting its intended deployment environment [8, 25]. The page size is fixed at 4 KB. Unless otherwise noted, experiments use a 4 GB cache with the prefix distribution and 100 threads, and TreePass uses an EWMA smoothing factor of $\alpha = 0.2$ for EI estimation. Following SplinterDB [8], each YCSB run measures throughput over 160 M operations (20 M for the scan-heavy YCSB-E) after loading the dataset and performing an 80 M-operation cache warm-up (10 M for YCSB-E). Each experiment is repeated three times, and we report the average.

Workloads. We evaluate with 673 M key-value pairs (24-byte keys, 100-byte values, totaling approximately 83.5 GB). We use a custom C++ benchmark that follows the YCSB [9] framework. Each workload is defined by a specific operation mix (insert / update / read / scan / delete / read-modify-write), and keys are drawn from a pre-generated key trace according to the target distribution.

We evaluate three access distributions: a Zipfian distribution ($\theta = 0.99$) to model skewed real-world workloads, a uniform distribution to evaluate behavior under non-skewed access, and a prefix distribution that models key-range locality

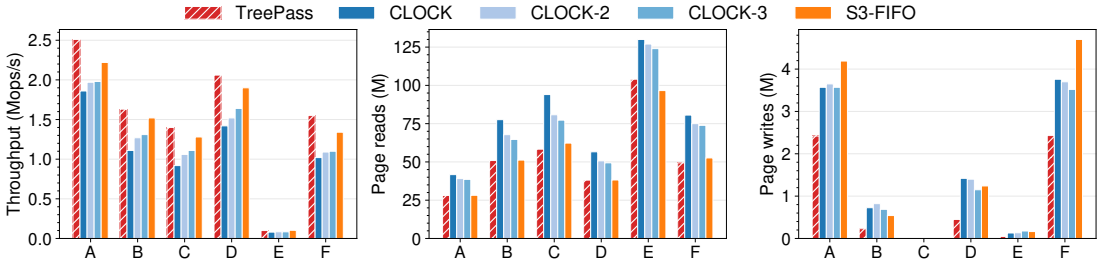


Figure 6: Throughput, page reads, and page writes across YCSB workloads (Zipfian 0.99)

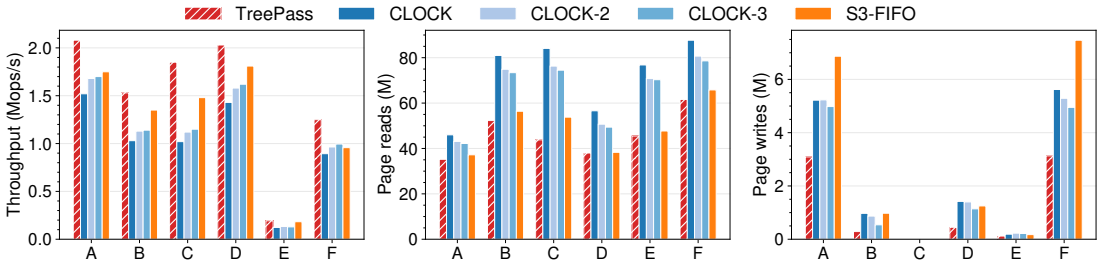


Figure 7: Throughput, page reads, and page writes across YCSB workloads (prefix)

observed in real-world KVS workloads [4], where accesses concentrate within a small number of hot key ranges while the remaining key space is rarely touched. We adopt the key-range access model from MixGraph [4], omitting other aspects such as variable value sizes and dynamic QPS patterns. We evaluate the standard YCSB workloads A–F: A (50% reads, 50% updates), B (95% reads, 5% updates), C (100% reads), D (95% reads of recently inserted keys, 5% inserts), E (95% range scans with length up to 100, 5% inserts), and F (50% reads, 50% read-modify-writes), under the Zipfian, uniform, and prefix distributions.

Baselines. We compare TreePass against the following four caching policies, all implemented on SplinterDB's ring-buffer cache:

- **CLOCK**: SplinterDB's original 1-bit CLOCK eviction policy that maintains a reference bit per entry and evicts entries whose bit is 0 when the hand passes.
- **CLOCK-2**: a 2-bit reference counter variant of CLOCK [31] that provides finer-grained eviction priority than the original 1-bit design.
- **CLOCK-3**: a 3-bit reference counter variant of CLOCK—the same counter width as TreePass, but without tree-structural signals.
- **S3-FIFO** [43]: a state-of-the-art eviction policy that uses three FIFO queues—a small queue that quickly evicts pages accessed only once (one-hit wonders), a main queue for frequently re-accessed pages, and a ghost queue that records recently evicted pages to improve future admission decisions. Each entry maintains a 2-bit reference counter within the queues. The ghost queue records metadata of recently evicted pages and is sized to hold 0.9× the number of entries in the cache; it is allocated separately from the cache memory, requiring additional memory proportional to the cache size.

4.2 YCSB Workloads

Figures 6 and 7 evaluate TreePass across YCSB A–F under Zipfian and prefix access distributions, respectively. TreePass achieves the highest throughput on all workloads under both distributions, except for YCSB-E under the Zipfian distribution.

Zipfian. Under Zipfian access, repeated accesses to hot keys repeatedly traverse the same root-to-leaf paths, creating subtree-level hotness for TreePass to exploit. Consequently, TreePass consistently outperforms CLOCK across YCSB A–F, improving throughput by 25%–52% while reducing page reads by 20%–39% (33% on average). On workloads with writes (A, B, D, E, and F), page writes are also reduced by 32%–68%, 53% fewer on average. CLOCK-2 and CLOCK-3 narrow this gap through finer-grained reference counting, but TreePass still outperforms them by 18%–42% across A–F, confirming that B^+-tree structural information provides gains beyond simply widening the counter.

Against S3-FIFO, TreePass achieves 7%–16% higher throughput. On write-heavy workloads (A and F), TreePass reduces page writes by 42% and 48%, respectively, directly cutting write I/O pressure. S3-FIFO generates the most page writes among all baselines. Originally designed for read-only caches [43], it does not distinguish clean from dirty pages, so dirty pages without prior access history (e.g., compaction output or updated pages) are quickly demoted and written back prematurely, even when they may soon be re-accessed.

Even on YCSB-C (100% reads), TreePass achieves 9% higher throughput while incurring fewer page reads. YCSB-E is the only workload where S3-FIFO outperforms TreePass by 3%, while also incurring fewer page reads.

Prefix. The prefix distribution models a key property of real-

Table 1: Tail latency under YCSB-A and YCSB-C

	YCSB-A		YCSB-C	
	P99	P99.9	P99	P99.9
TreePass	**495 µs**	**941 µs**	**415 µs**	**594 µs**
S3-FIFO	514 µs	1.04 ms	491 µs	678 µs
CLOCK-3	547 µs	1.03 ms	521 µs	727 µs
CLOCK-2	560 µs	1.07 ms	527 µs	736 µs
CLOCK	579 µs	1.11 ms	542 µs	759 µs

world KVS workloads— accesses concentrated within a small number of hot key ranges [4]. Hot key ranges directly correspond to hot subtrees in the B^+-tree, allowing TreePass's compaction counter and access counter to accurately identify which pages are worth retaining. TreePass outperforms CLOCK by 37%–81% across A–F in throughput, with 23%–48% fewer page reads and 37%–70% fewer page writes on workloads with writes (A, B, D, E, and F), a larger gap than under Zipfian. Against S3-FIFO, TreePass outperforms across all six workloads by 5%–31%, with an average advantage of 18%. TreePass also reduces page writes by 31%–70%, 56% on average, relative to S3-FIFO on workloads with writes.

Uniform. Uniform workloads inherently exhibit much less locality than skewed workloads, making caching less effective regardless of eviction policy [28]. However, even under uniform key access, upper-level B^+-tree nodes cover wider key ranges and are visited on every traversal. As a result, accesses naturally accumulate toward higher levels of the tree, creating a height-dependent hotness pattern that TreePass can still exploit. TreePass still outperforms CLOCK by 31%–45% across A–F, but the gap against S3-FIFO narrows to 2%–9%. On write-heavy workloads (A and F), TreePass reduces page writes by 28%–36% relative to CLOCK, 34%–36% relative to CLOCK-2 and CLOCK-3, and 35%–36% relative to S3-FIFO.

These results from the three distributions confirm that the subtree-level signal, while most effective when key-range locality is strong, is not tied to a particular access distribution. Across all three distributions, TreePass consistently reduces both page reads and page writes. Page reads directly reduce cache miss latency, whereas page writes improve throughput indirectly by reducing I/O contention with foreground reads. In these YCSB workloads, page writes account for only a small fraction of total page I/O (5% on average for workloads A, B, D, and F), so the throughput gains are dominated by the reduction in page reads. As the value size increases, page writes become more significant (21% for 1024-byte values), making write reduction more beneficial (§4.6).

4.3 Tail Latency

Table 1 reports P99 and P99.9 latency under YCSB-A and YCSB-C. TreePass achieves the lowest tail latency across both workloads and all percentiles. The improvement is more pronounced under YCSB-C (100% reads): TreePass reduces P99 by 15% relative to S3-FIFO and 23% relative to CLOCK, with P99.9 reductions of 12% and 22%, respectively. In a read-only workload, tail latency is determined almost entirely by the frequency and cost of cache misses; TreePass's higher hit ratio translates directly into fewer disk reads on the tail of the latency distribution. Under YCSB-A (50% reads, 50% updates), TreePass reduces P99 by 4%–15% and P99.9 by 9%–15% relative to baselines. The gap is narrower than under YCSB-C because update operations contribute additional latency variance from write path processing that is independent of cache eviction quality.

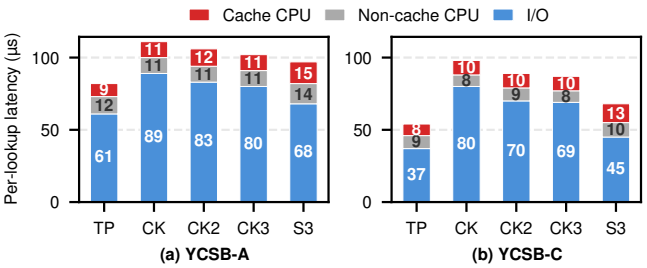


Figure 8: Per-lookup latency breakdown

4.4 Latency Breakdown

To understand the source of TreePass's throughput advantage, we decompose average per-lookup latency into three components: I/O time (disk read stalls on cache misses), cache CPU time (cache-internal operations), and non-cache CPU time (the KVS's own operations). Figure 8 shows the results for YCSB-A and YCSB-C, where a number in the bars is the latency of each component, and TP, CK, CK2, CK3, and S3 denote TreePass, CLOCK, CLOCK-2, CLOCK-3, and S3-FIFO, respectively. The primary contributor to TreePass's advantage is lower I/O time. Under YCSB-A, TreePass spends 61.2 µs on I/O per lookup, compared to 68.1 µs for S3-FIFO and 89.0 µs for CLOCK, reflecting fewer cache misses due to higher-quality eviction decisions. Under YCSB-C, the difference is even larger: TreePass's I/O time (37.3 µs) is 54% lower than CLOCK (80.3 µs) and 16% lower than S3-FIFO (44.6 µs). Despite maintaining per-pivot metadata (AC and CC), TreePass's cache CPU time (8.9 µs under YCSB-A, 7.9 µs under YCSB-C) is comparable to CLOCK (11.4 µs, 9.8 µs) and lower than S3-FIFO (15.0 µs, 12.9 µs). S3-FIFO's higher cache CPU time reflects the cost of managing its three-queue structure and ghost queue lookups on every cache miss. TreePass avoids this overhead by embedding structural signals directly in existing pivot entries rather than maintaining a separate data structure.

4.5 Ablation Study

To isolate the contribution of each component, we incrementally enable *differentiated initialization* (DI) and *early eviction* (EE) on top of CLOCK-3, which serves as the baseline—it uses the same 3-bit reference counter as TreePass but without any tree-structural signals. DI assigns differentiated initial reference counters based on page context: clean pages loaded from disk receive 0, dirty pages receive 1, and compaction output pages receive a value determined by both the reference

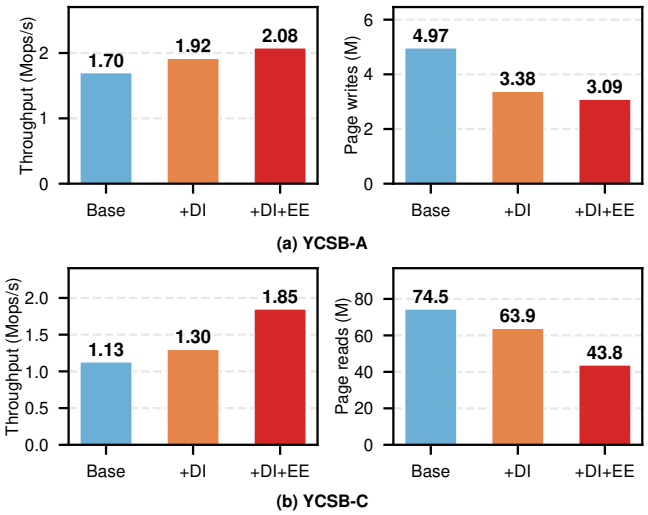


Figure 9: Ablation study

counters of their input pages and the CI/EI comparison (§3). Pages created by copy-on-write or node split derive their initial reference-counter values from their predecessors. Figure 9 shows the results.

Differentiated initialization. Adding DI alone improves throughput by 13% and 15% under YCSB-A and YCSB-C, respectively. Under YCSB-A, the primary effect is a 32% reduction in page writes: DI prevents unnecessary writeback of dirty pages that are likely to be invalidated before eviction. Page reads also decrease by 11% because assigning 0 to clean pages allows them to be evicted in a single sweep if not re-accessed, freeing cache space for pages that are more likely to be reused. Under YCSB-C, where no page writes occur, the same mechanism reduces page reads by 14%.

Early eviction. Adding EE on top of DI yields a further 8% throughput gain under YCSB-A and a substantially larger 42% gain under YCSB-C. EE identifies pages unlikely to be re-accessed before eviction, removing them proactively during branch tree traversal. The effect is particularly strong under YCSB-C: since no write I/O occurs, throughput depends entirely on cache hit quality, and EE's ability to reclaim cache space from cold pages reduces page reads by an additional 32% beyond DI alone.

4.6 Impact of Value Size

To examine the impact of value size on TreePass's performance, we evaluate with 24-byte keys and 1024-byte values, following the configuration used in prior SplinterDB-based evaluations [7,8], with 84 M load operations. We use warm-up and measurement phases of 20 M and 40 M operations, scaled down from the 100-byte setting used in §4.2. The resulting dataset is approximately 88 GB, comparable to the 83.5 GB dataset used in that setting. Figure 10 shows the results.

TreePass outperforms all baselines on all six workloads, with throughput improvements of 16%–83% over CLOCK and 10%–101% over S3-FIFO. Compared with the 100-byte setting (Figure 7), the most notable change is that TreePass's throughput advantage over S3-FIFO on write-heavy workloads expands substantially: from 19% to 79% on YCSB-A, and from 31% to 101% on YCSB-F. As analyzed in the ablation study (§4.5), TreePass leverages tree-structural signals to differentiate the initial reference counter of newly admitted pages, suppressing unnecessary writebacks. A large-value setting amplifies the effect of this writeback suppression: due to the page-granularity I/O model, read cost is relatively insensitive to value size, whereas write cost grows because fewer KV pairs are packed into each page. Indeed, on workloads with writes (A, B, D, and F), the average fraction of write I/O within total page I/O across all policies increases from 5% in the 100-byte setting to 21% in the 1024-byte setting—roughly a fourfold increase. In this regime, on write-heavy workloads (A and F), TreePass issues 46%–53% fewer page writes than S3-FIFO, making the performance gap more pronounced than in the 100-byte setting. On read-heavy workloads (B, C, D, and E) the gap (10%–23%) is similar to that in the 100-byte setting (5%–25%).

4.7 Cache Size Sensitivity

Figure 11 shows throughput as the cache size varies from 1 GB to 8 GB (1.2%–9.6% of the 83.5 GB dataset) under YCSB-A and YCSB-C with the prefix distribution. TreePass outperforms all baselines across all cache sizes under both workloads. Under YCSB-A, TreePass maintains a consistent 16%–19% throughput advantage over S3-FIFO and 27%–37% over CLOCK at 1–4 GB. At 8 GB, where most of the hot working set fits in cache, eviction pressure decreases and the gap narrows: TreePass leads S3-FIFO by 17% and CLOCK by 9%. Across all cache sizes, TreePass generates 53%–55% and 39%–52% fewer page writes than S3-FIFO and CLOCK, respectively, under YCSB-A, reducing I/O contention on the write path. Notably, at 8 GB, S3-FIFO achieves the lowest throughput among all five policies despite having similar hit ratios. At large cache sizes, eviction pressure is low and most pages are retained regardless of the eviction policy; the ghost queue then adds management overhead without providing proportional benefit.

Under YCSB-C, the throughput gap varies more widely with cache size. At 1 GB, all policies are heavily I/O-bound and the difference between TreePass and S3-FIFO is modest (4%). The gap peaks at 4 GB (25% over S3-FIFO, 81% over CLOCK), where TreePass's eviction quality has the greatest impact. At 8 GB, the gap decreases to 13% over S3-FIFO as hit ratios for all policies converge above 99%.

4.8 Real-World Workloads

To evaluate TreePass under real-world access patterns, we replay a production trace from Twitter's in-memory key-value cache [42] on SplinterDB. The released collection contains 54 anonymized clusters with diverse access patterns and operation mixes; we select Cluster-14, whose dataset is much larger than the cache sizes we evaluate. It consists of 65% reads, 13% updates, and 22% deletes, with approximately 50 GB

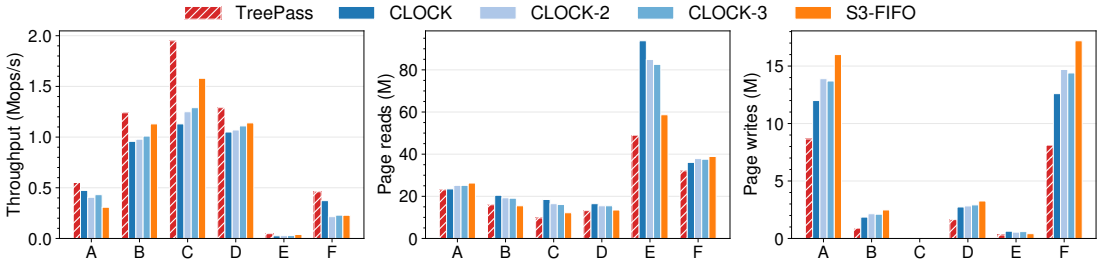


Figure 10: Throughput, page reads, and page writes across YCSB workloads with 1024-byte values (prefix)

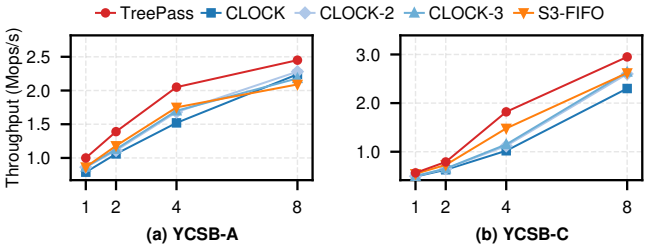


Figure 11: Throughput as a function of cache size

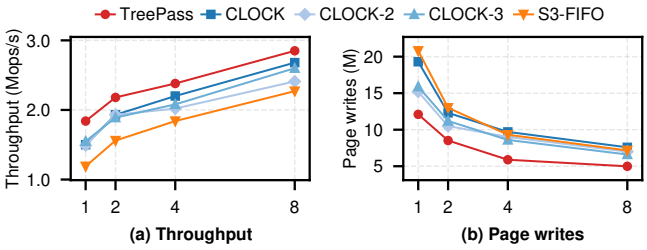


Figure 12: Performance on the Twitter trace

of key-value pairs loaded before replay. We bulk-load each unique key-value pair in first-occurrence order, then replay the operations in trace order, preserving each pair's original key and value sizes while capping the small fraction of values that exceed SplinterDB's per-pair size limit. Following §4.1, we perform an 80 M-operation warm-up and then measure over 160 M operations. We report throughput and page writes as a function of cache size (1–8 GB), as in §4.7, to place the cache under memory pressure comparable to the YCSB setting.

Figure 12 shows that TreePass outperforms all baselines across all cache sizes. TreePass improves throughput by 6%–23% and 26%–55% over CLOCK and S3-FIFO, respectively. The margin is largest at the smallest cache and narrows at 8 GB as eviction pressure decreases. Across cache sizes, TreePass reduces page reads by 6%–27% and 7%–25%, and page writes by 31%–39% and 30%–42%, over CLOCK and S3-FIFO, respectively. These results confirm that TreePass's benefits carry over from synthetic YCSB workloads to a real-world production workload.

4.9 Parameter Sensitivity

Early eviction threshold. Early eviction classifies a page as cold when its reuse interval exceeds the eviction interval (RI > EI). To study the sensitivity of this threshold, we

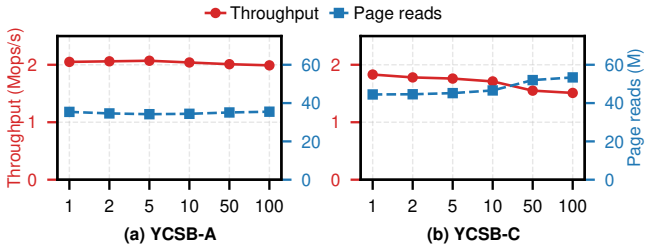


Figure 13: Effects of the EI multiplier

introduce a multiplier m that relaxes the criterion to RI > $m\times$ EI, making early eviction less aggressive as m increases. Figure 13 shows throughput and page reads under YCSB-A and YCSB-C as m varies from 1 to 100. Under YCSB-A, throughput is largely insensitive to m: varying from 1 to 100 changes throughput by only 3% (2.08 Mops/s to 2.02 Mops/s). Dirty pages produced by writes are protected by differentiated initialization, leaving little room for the eviction threshold to affect overall throughput. Under YCSB-C, the effect is stronger. Performance degrades gracefully up to $m = 10$ (5% throughput decrease relative to $m = 1$), but drops more sharply beyond that point: at $m = 100$, throughput decreases by 16% as page reads increase from 43.8M to 53.1M. In a read-only workload, all cached pages are clean, so the eviction threshold directly affects how quickly cold pages are reclaimed. As a result, the default setting of $m = 1$ provides the best throughput across both workloads.

EI estimation parameters. We also evaluate TreePass's sensitivity to the two parameters of EI estimation: the EWMA smoothing factor α and the initial EI value. We vary α from 0.1 to 0.9 (in steps of 0.1) and the initial EI across six values spanning a 40$\times$ range (1–40 s). Across all 54 combinations, throughput varies by at most 2.0% on YCSB-A and 2.2% on YCSB-C, confirming that TreePass is insensitive to both parameters.

5 Discussion

5.1 Generality

TreePass targets larger-than-memory B^+-tree-based storage systems with a user-level page cache; SplinterDB is simply our evaluated instance. It exploits key-range locality through its subtree-level hotness signal, while handling out-of-place updates through two complementary mechanisms: hotness preservation and writeback suppression. We next discuss how

these mechanisms generalize to a broader class of systems.

Subtree-level hotness signal. This mechanism only requires parent-to-child traversal and a user-level page cache, so it applies to B^+-tree-based storage systems regardless of whether updates are in-place or out-of-place—e.g., B^+-tree storage engines such as LeanStore [25] and WiredTiger [30]. For LSM-trees, lookups search multiple SSTables across levels rather than following a single parent-to-child hierarchy, so the signal does not transfer directly. Within a single SSTable, however, the index block maps to data blocks in a parent-to-child structure, so an access counter at the index block could be used to estimate the RI of data blocks within that SSTable. Cross-SSTable signals remain unavailable.

Hotness preservation. When splits or page rewrites move a page to a new location, TreePass inherits the predecessor's reference counter (or distributes it evenly across split pages), preserving access history. The same principle applies broadly to other systems that use out-of-place updates, such as WiredTiger [30]. More generally, any CoW-based B^+-tree can preserve hotness across address changes by transferring the predecessor's counter to the rewritten page.

Write suppression. In systems where a user-level page cache controls the writeback timing of dirty pages, the dirty-page generation rate can be used to predict pages likely to be invalidated soon and avoid unnecessary writebacks. Systems that perform compaction (e.g., SplinterDB) use CI as this rate; for systems without compaction, such as in-place update KVSs or CoW-based B^+-trees, the same principle can be applied to the update rate to defer writeback of dirty pages likely to be updated again soon.

Note that as future work, TreePass can be extended to a cache shared across multiple instances. The subtree signal is maintained independently in each tree's pivot entries, while EI is shared and updated atomically across instances.

5.2 Durability and Crash Consistency

SplinterDB supports write-ahead logging (WAL); when enabled, committed updates remain durable and consistency is preserved through WAL replay after crashes. TreePass does not modify the WAL path, so this guarantee is unchanged; our experiments disable WAL, as in prior storage-system evaluations where logging is orthogonal to the proposed mechanism [16, 17, 21, 44]. TreePass's AC, CC, and EI are maintained only in memory and are reset after a crash. After a crash, the cache follows the same initialization procedure as in §3.1: it operates as standard CLOCK until the first CLOCK sweep completes, after which EI is initialized and early eviction is enabled.

6 Related Work

B^+-tree-based storage systems. Representative B^+-tree storage systems include SplinterDB [7, 8], LeanStore [25], WiredTiger [30], TreeLine [44], and Bf-Tree [17]. Systems with frequent out-of-place updates (e.g., SplinterDB and WiredTiger) lose accumulated access history when pages are rewritten. TreePass improves cache eviction in this setting, exploiting structural signals already present in the B^+-tree.

General-purpose cache eviction policies. General-purpose cache eviction policies include CLOCK [10], GCLOCK [31], CLOCK-Pro [18], 2Q [20], LIRS [19], ARC [29], TinyLFU [12], SIEVE [45], S3-FIFO [43], and learned policies such as LeCaR [37], CACHEUS [35], GL-Cache [40], and 3L-Cache [46]. These policies associate history with individual pages, whereas TreePass exploits B^+-tree structure to derive address-independent eviction signals without additional metadata.

Tree-aware caching. Several systems incorporate tree structure into buffer management, though with different goals than TreePass. LeanStore [25] targets larger-than-memory B^+-trees for NVMe-array environments and evicts inner pages only after their children. Its cache replacement relies on random sampling, whereas TreePass derives quantitative eviction signals from pivot accesses to guide page-level eviction. Unlike LeanStore based on in-place updates, TreePass also preserves hotness across out-of-place updates and suppresses unnecessary writebacks. DEX [28] constrains admission and eviction using parent–child relationships to reduce remote-memory accesses, but does not exploit subtree hotness.

Hotness continuity across out-of-place updates. The loss of access history when data is rewritten has been studied in LSM-tree systems. ElasticBF [26], Mnemosyne [47], and HotRAP [33] track or estimate hotness at the SSTable, block, or record level. In contrast, TreePass preserves hotness across out-of-place updates in a B^+-tree page cache, which to the best of our knowledge has not been explored previously.

7 Conclusion

In this paper, we presented TreePass, a lightweight CLOCK-based caching system that exploits subtree-level signals to improve cache management. TreePass achieves up to 101% higher throughput than state-of-the-art caching policies.

Acknowledgments

We thank our shepherd, Patrick P. C. Lee, and the anonymous reviewers for their valuable feedback. This work was partly supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP)-ITRC (Information Technology Research Center) grant funded by the Korea government (MSIT) (IITP-2026-RS-2021-II211817, 20%), Institute of Information & Communications Technology Planning & Evaluation (IITP) grants funded by the Korea government (MSIT) (No. RS-2025-02214652, Development of SoC Technology for AI Semiconductor-Converged Pooled Storage/Memory, 40%; No. RS-2024-00459026, Energy-Aware Operating System for Disaggregated System, 40%), and Samsung Electronics Co., Ltd. Young-ri Choi is the corresponding author.

References

[1] Jung-Sang Ahn, Mohiuddin Abdul Qader, Woon-Hak Kang, Hieu Nguyen, Guogen Zhang, and Sami Ben-Romdhane. Jungle: Towards dynamically adjustable key-value store by combining LSM-tree and Copy-On-Write B+-tree. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[2] Jung-Sang Ahn, Chiyoung Seo, Ravi Mayuram, Rahim Yaseen, Jin-Soo Kim, and Seungryoul Maeng. ForestDB: A fast key-value storage system for variable-length string keys. *IEEE Transactions on Computers*, 65(3):902–915, 2016.

[3] Minwoo Ahn, Jeongmin Han, Youngjin Kwon, and Jinkyu Jeong. Identifying On-/Off-CPU bottlenecks together with blocked samples. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 893–910, 2024.

[4] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, 2020.

[5] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2):Article 4, 2008.

[6] Howard Chu. MDB: A memory-mapped database and backend for OpenLDAP. In *Proceedings of the 3rd International Conference on LDAP (LDAPCon 11)*, 2011.

[7] Alex Conway, Martín Farach-Colton, and Rob Johnson. SplinterDB and Maplets: Improving the tradeoffs in key-value store compaction policy. *Proceedings of the ACM on Management of Data*, 1(1):Article 46, 2023.

[8] Alex Conway, Abhishek Gupta, Vijay Chidambaran, Martin Farach-Colton, Rick Spillane, Amy Tai, and Rob Johnson. SplinterDB: closing the bandwidth gap for NVMe key-value stores. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 49–63, 2020.

[9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC 10)*, pages 143–154, 2010.

[10] Fernando J. Corbató. *A Paging Experiment with the Multics System*. Massachusetts Institute of Technology, 1968.

[11] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP 07)*, pages 205–220, 2007.

[12] Gil Einziger, Roy Friedman, and Ben Manes. TinyLFU: A highly efficient cache admission policy. *ACM Transactions on Storage*, 13(4):Article 35, 2017.

[13] Facebook RocksDB Team. RocksDB: A persistent key-value store for fast storage environments. https://rocksdb.org/, 2022. Accessed July 13, 2026.

[14] Guanyu Feng, Huanqi Cao, Xiaowei Zhu, Bowen Yu, Yuanwei Wang, Zixuan Ma, Shengqi Chen, and Wenguang Chen. TriCache: A User-Transparent block cache enabling High-Performance Out-of-Core processing with In-Memory programs. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 395–411, 2022.

[15] Brad Fitzpatrick. Distributed caching with Memcached. *Linux Journal*, 2004(124):5, 2004.

[16] Gabriel Haas and Viktor Leis. What modern NVMe storage can do, and how to exploit it: High-performance I/O for high-performance storage engines. *Proceedings of the VLDB Endowment*, 16(9):2090–2102, 2023.

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

[18] Song Jiang, Feng Chen, and Xiaodong Zhang. CLOCK-Pro: An effective improvement of the CLOCK replacement. In *Proceedings of the 2005 USENIX Annual Technical Conference (USENIX ATC 05)*, pages 323–336, 2005.

[19] Song Jiang and Xiaodong Zhang. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 02)*, pages 31–42, 2002.

[20] Theodore Johnson and Dennis Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB 94)*, pages 439–450, 1994.

[21] Konstantinos Kanellis, Badrish Chandramouli, Ted Hart, and Shivaram Venkataraman. From FASTER to F2: Evolving concurrent key-value store designs for large skewed workloads. *Proceedings of the VLDB Endowment*, 18(12):4910–4923, 2025.

[22] Wonbae Kim, Chanyeol Park, Dongui Kim, Hyeongjun Park, Young-ri Choi, Alan Sussman, and Beomseok Nam. ListDB: Union of Write-Ahead logs and persistent SkipLists for incremental checkpointing on persistent memory. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 161–177, 2022.

[23] Jongbaeg Lee, Gihwan Oh, and Sang-Won Lee. Boosting compaction in B-tree based key-value store by exploiting parallel reads in flash SSDs. *IEEE Access*, 9:56344–56353, 2021.

[24] Jongsung Lee, Donguk Kim, and Jae W. Lee. WALTZ: Leveraging zone append to tighten the tail latency of LSM-tree on ZNS SSD. *Proceedings of the VLDB Endowment*, 16(11):2884–2896, 2023.

[25] Viktor Leis. LeanStore: A high-performance storage engine for NVMe SSDs. *Proceedings of the VLDB Endowment*, 17(12):4536–4545, 2024.

[26] Yongkun Li, Chengjin Tian, Fan Guo, Cheng Li, and Yinlong Xu. ElasticBF: Elastic bloom filter with hotness awareness for boosting read performance in large key-value stores. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 739–752, 2019.

[27] Zhiyue Li and Guangyan Zhang. StreamCache: Revisiting page cache for file scanning on fast storage devices. In *Proceedings of the 2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 1119–1134, 2024.

[28] Baotong Lu, Kaisong Huang, Chieh-Jan Mike Liang, Tianzheng Wang, and Eric Lo. DEX: Scalable range indexing on disaggregated memory. *Proceedings of the VLDB Endowment*, 17(10):2603–2616, 2024.

[29] Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST 03)*, pages 115–130, 2003.

[30] MongoDB, Inc. WiredTiger storage engine. https://www.mongodb.com/docs/manual/core/wiredtiger/, 2021. Accessed July 13, 2026.

[31] Victor F. Nicola, Asit Dan, and Daniel M. Dias. Analysis of the generalized clock buffer replacement scheme for database transaction processing. *ACM SIGMETRICS Performance Evaluation Review*, 20(1):35–46, 1992.

[32] Yifan Qiao, Xubin Chen, Ning Zheng, Jiangpeng Li, Yang Liu, and Tong Zhang. Closing the B+-tree vs. LSM-tree write amplification gap on modern storage hardware with built-in transparent compression. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 69–82, 2022.

[33] Jiansheng Qiu, Fangzhou Yuan, Mingyu Gao, and Huanchen Zhang. HotRAP: hot record retention and promotion for LSM-trees with tiered storage. In *Proceedings of the 2025 USENIX Annual Technical Conference (USENIX ATC 25)*, pages 497–511, 2025.

[34] Ohad Rodeh. B-trees, shadowing, and clones. *ACM Transactions on Storage*, 3(4):Article 2, 2008.

[35] Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning cache replacement with CACHEUS. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 341–354, 2021.

[36] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. Learning relaxed Belady for content distribution network caching. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 529–544, 2020.

[37] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving cache replacement with ML-based LeCaR. In *Proceedings of the 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[38] VMware. SplinterDB. https://github.com/vmware/splinterdb/commit/2b6d030f7d7aad1dd9c5cbc2e096c0548b5e959c, 2025. Accessed July 13, 2026.

[39] Xiaoliang Wang, Peiquan Jin, Yongping Luo, and Zhaole Chu. Range Cache: An Efficient Cache Component for Accelerating Range Queries on LSM-based key-value stores. In *Proceedings of the 2024 IEEE 40th International Conference on Data Engineering (ICDE 24)*, pages 488–500, 2024.

[40] Juncheng Yang, Ziming Mao, Yao Yue, and K. V. Rashmi. GL-Cache: Group-level learning for efficient and high-performance caching. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 115–134, 2023.

[41] Juncheng Yang, Ziyue Qiu, Yazhuo Zhang, Yao Yue, and K. V. Rashmi. FIFO can be better than LRU: the power

of lazy promotion and quick demotion. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HotOS 23)*, pages 70–79, 2023.

[42] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208, 2020.

[43] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and K. V. Rashmi. FIFO queues are all you need for cache eviction. In *Proceedings of the 29th ACM SIGOPS Symposium on Operating Systems Principles (SOSP 23)*, pages 130–149, 2023.

[44] Geoffrey X. Yu, Markos Markakis, Andreas Kipf, Per-Åke Larson, Umar Farooq Minhas, and Tim Kraska. TreeLine: an update-in-place key-value store for modern storage. *Proceedings of the VLDB Endowment*, 16(1):99–112, 2022.

[45] Yazhuo Zhang, Juncheng Yang, Yao Yue, Ymir Vigfusson, and K. V. Rashmi. SIEVE is simpler than LRU: an efficient Turn-Key eviction algorithm for web caches. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1229–1246, 2024.

[46] Wenbin Zhou, Zhixiong Niu, Yongqiang Xiong, Juan Fang, and Qian Wang. 3L-Cache: Low overhead and precise learning-based eviction policy for caches. In *Proceedings of the 23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 237–254, 2025.

[47] Zichen Zhu, Yanpeng Wei, Ju Hyoung Mun, and Manos Athanassoulis. Mnemosyne: Dynamic workload-aware BF tuning via accurate statistics in LSM trees. *Proceedings of the ACM on Management of Data*, 3(3):Article 190, 2025.

A Artifact Appendix

Abstract

The artifact contains the full implementation of TreePass together with the components needed to build and benchmark it: a TreePass-extended fork of SplinterDB whose cache backend embeds per-pivot Tree Counters in B^+-tree pivot entries and uses them to drive eviction, a YCSB-style benchmark driver, a self-contained smoke test, and the driver used for the paper-scale experiments. The artifact was evaluated by the FAST '27 Artifact Evaluation Committee.

Scope

The artifact allows researchers to validate that all TreePass mechanisms described in the paper—per-pivot Access and Compaction Counters, early eviction during branch-tree traversal, and differentiated reference-counter initialization for pages created by out-of-place updates—are implemented in SplinterDB and can be exercised end-to-end. The included smoke test is a scaled-down, functional version of the YCSB experiments in Section 4.2 (Figures 6 and 7). On hardware comparable to the paper's setup (Section 4.1), the paper-scale driver reproduces TreePass's YCSB results, including the value-size and cache-size configurations (Sections 4.6 and 4.7). Note that the artifact ships only the TreePass cache backend: the baseline eviction policies compared in the paper (the CLOCK variants and S3-FIFO) must be ported separately, and analyses requiring additional instrumentation or external traces (e.g., the tail-latency, ablation, and Twitter-trace experiments) are outside the artifact's scope. Beyond validation, the artifact can be reused as a drop-in caching backend for SplinterDB and as a basis for research on tree-aware cache management.

Contents

The repository contains (1) `splinterdb/`, the TreePass-extended SplinterDB fork—TreePass itself lives in `src/treepass.{c,h}`, derived from the upstream `clockcache`, with `btree.c`, `trunk.c`, `core.c`, and related files modified to embed Tree Counters in pivot entries and route them through eviction; (2) `bench/`, a YCSB-style benchmark driver supporting the prefix, Zipfian, and uniform query distributions with warm-up/measurement phase splits; (3) `examples/run_smoke.py`, a self-contained smoke test, with `gen_smoke_load.py` generating its synthetic dataset; and (4) `ycsb.py`, the driver for the paper-scale experiments, together with instructions for generating the paper's datasets.

Hosting

The artifact is hosted on GitHub at `https://github.com/unist-ssl/TreePass` (branch `main`, tag `fast27`, commit `41224ad`) and is permanently archived on Zenodo (DOI: 10.5281/zenodo.21591527). The code is licensed under Apache-2.0, inherited from SplinterDB.

Requirements

The artifact targets Ubuntu 22.04 (tested on 22.04.2 LTS, kernel 5.15, glibc 2.35; older glibc such as Ubuntu 18.04's 2.27 lacks the `gettid()` wrapper and is not supported) with `build-essential`, `cmake` (3.17+), `python3` (3.7+), and the `libaio-dev`, `libxxhash-dev`, and `libgflags-dev` packages. The smoke test runs on any commodity x86-64 machine with roughly 2 GB of free RAM and a few hundred MB of free disk; no root privileges or external datasets are required. The paper-scale driver additionally requires `numactl` and `cpupower`, which it uses to interleave memory allocation

across sockets and to pin the CPU frequency governor. Reproducing the paper's absolute numbers requires hardware comparable to the setup in Section 4.1: a dual-socket Intel Xeon Gold 6526Y server (32 cores, 64 hardware threads) with 314 GB of DDR5 DRAM and a Samsung PM9A3 1.92 TB NVMe SSD (PCIe 4.0 $\times$4).

Getting Started

After installing the packages above and cloning the repository, running `python3 examples/run_smoke.py` builds the TreePass library and the benchmark, generates a synthetic 1 M-key dataset, loads it once, and runs YCSB-A, B, C, and F under the uniform, Zipfian, and prefix distributions (12 timed configurations). Each configuration writes one log under `result/smoke_<timestamp>/`. A query configuration finishes cleanly if its log ends with the `TreePass bench done` banner and contains a summary line reporting the total throughput, which appears a few lines before the banner. Both are present regardless of the settings. In addition, since the smoke runner enables `--use_stats`, each log also carries per-page-type cache hit-ratio statistics (omitted when statistics are disabled). Thread count, cache sizes, operations per phase, database path, and build parallelism are all exposed as command-line flags. The paper-scale experiments are driven by `ycsb.py` with user-supplied dataset CSVs, as documented in the repository README.